

AD-A069 299

AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OHIO SCH--ETC F/G 9/2
IMPLEMENTATION AND TESTING OF NUMERICAL ANALYSIS TECHNIQUES IN --ETC(U)
MAR 79 R A ADAMS

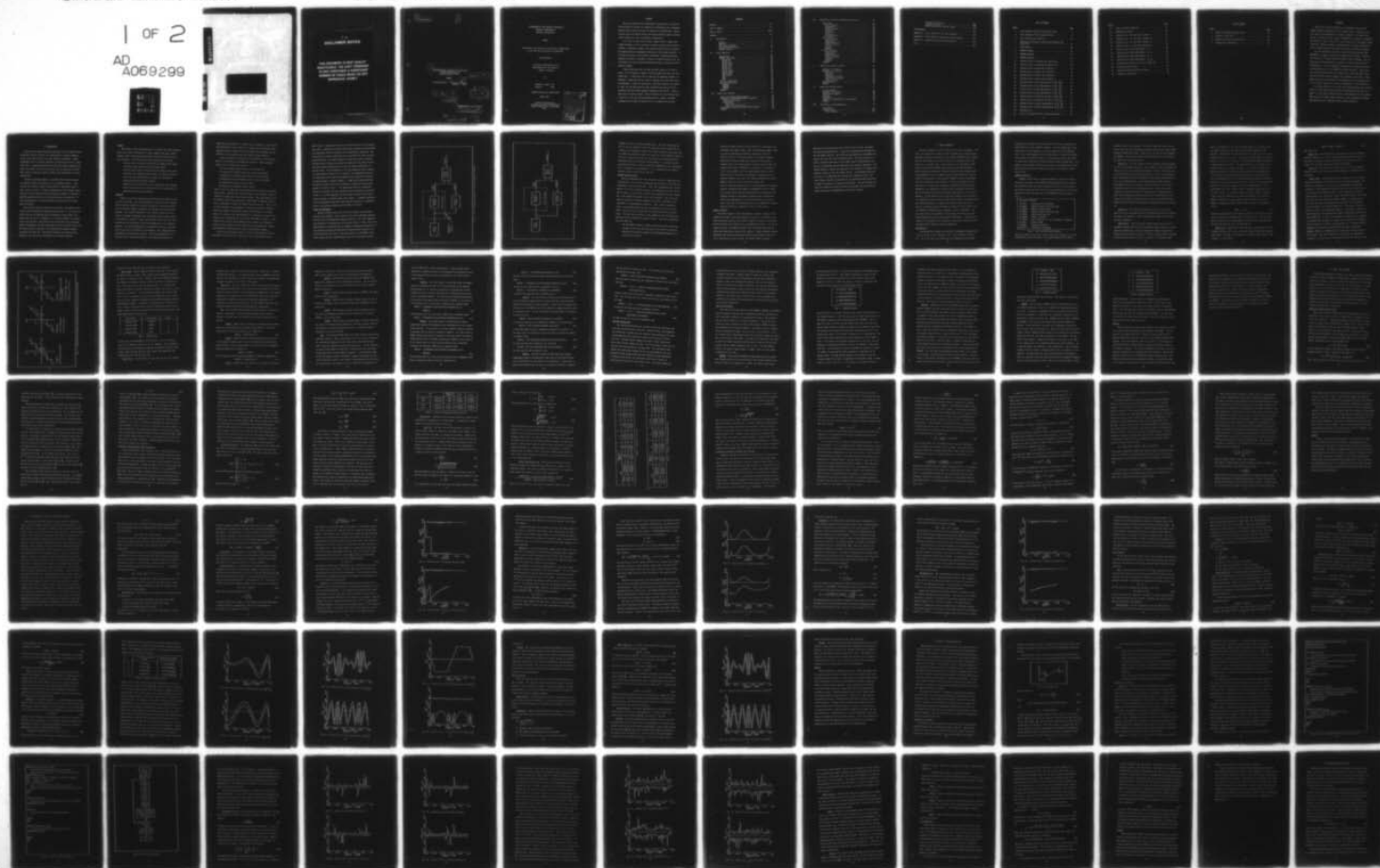
UNCLASSIFIED

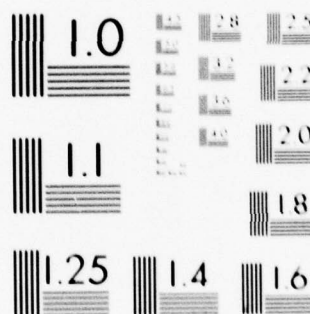
AFIT/GCS/EE/79-1

NL

1 OF 2

AD
A069299





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A



DISCLAIMER NOTICE

**THIS DOCUMENT IS BEST QUALITY
PRACTICABLE. THE COPY FURNISHED
TO DDC CONTAINED A SIGNIFICANT
NUMBER OF PAGES WHICH DO NOT
REPRODUCE LEGIBLY.**

14

AFIT/GCS/EE/79-1

1

IMPLEMENTATION AND TESTING OF NUMERICAL
ANALYSIS TECHNIQUES IN
AVIONICS APPLICATIONS.

THESIS

AFIT/GCS/EE/79-1

Richard A. Adams
Captain USAF

9 Master's thesis

11 Mar 79

12 255p

DDC
RECEIVED
JUN 4 1979
A

DISTRIBUTION STATEMENT A

Approved for public release
Distribution Unlimited

79 05 30 235
012 225 LB

IMPLEMENTATION AND TESTING OF NUMERICAL
ANALYSIS TECHNIQUES IN
AVIONICS APPLICATIONS

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology

Air University

in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

by

Richard A. Adams, B.S.
Captain USAF

Graduate Electrical Engineering

March 1979

School of Engineering
Air Force Institute of Technology
Wright-Patterson Air Force Base,
Ohio

Accession For	
DTIC	DTIC
DDC TAB	
Unannounced	
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or special
A/2344	

Preface

The n-bit simulation tool developed by Captain Gary A. Klein has been modified to include the capability of simulating two's-complement machines which truncate results, as opposed to rounding them. Anyone who uses this tool should realize the effect different number representation schemes can have on arithmetic computations.

It is easy to state that we need a computer with a longer word-length whenever we fail to achieve a required accuracy from a piece of software. Sometimes, however, this required accuracy can be obtained by employing better programming techniques on the original machine. In writing this thesis, I am primarily interested in applying numerical analysis techniques to software to obtain a required accuracy with the constraints of a given fixed wordlength and number representation and handling schemes.

I would especially like to thank my thesis advisor, Dr. Gary B. Lamont, for his guidance, support, and many suggestions when they were needed most. I would also like to thank my two committee members, Dr. Thaddeus L. Regulinski and Dr. Peter S. Maybeck, for their advice and encouragement. A special thanks goes to my thesis sponsor, Dr. Donald L. Moon, for the many hours he spent helping me get started and for providing some much needed guidance throughout this period. Lastly, I would like to thank my typist, Cheryl Gilliland, for her diligent work. I thank all of you for your continued patience, support, and understanding that has made it possible for me to complete this thesis.

Contents

Preface	ii
List of Figures	vi
List of Tables	viii
Abstract	ix
I. Introduction	1
Purpose	2
Approach	2
Tools and Techniques	4
Assumptions/Constraints	7
Chapter Synopsis	8
II. N-bit Simulator	10
Preprocessor	10
SETNBIT Subroutine	11
Option One	12
Option Two	12
Option Three	12
Option Four	13
Option Five	14
Option Six	14
Option Seven	14
Option Eight	16
Option Nine	16
Option Ten	17
KEY	17
TKEY	18
Function Subroutines	21
Special Subroutines	22
ONETRNC	22
TWOETRNC	24
ROUNDER	25
Summary	26
III. Monte Carlo Testing	28
Generating Pseudo-Random Samples	28
Testing of Pseudo-Random Number Generators	30
Uniform Distribution	31
Independence	33
Runs Test	33
Serial Correlation Test	34
Termination Criteria for Monte Carlo Testing	36
Summary	42

IV.	Evaluation of Avionics Mathematical Routines	43
	Square Root	44
	Specifications	44
	Assumptions	45
	Testing Criteria	45
	Solution 1	46
	Solution 2	48
	Solution 3	51
	Recommendations	52
	Sine Function	54
	Specifications	54
	Assumptions	55
	Range Reductions	55
	Testing Criteria	56
	Solution	57
	Summary	62
	Cosine Function	62
	Specifications	62
	Assumptions	62
	Range Reductions	63
	Testing Criteria	63
	Solution	63
	Summary	65
	Summary	65
V.	Analysis of Avionics Routine	66
	Bearing to Go Routine	66
	Bearing to Go Analysis	68
	Objectives	68
	Approach	68
	Termination Criteria	73
	General Plot Analysis	74
	Spike Analysis	80
	Recommendations	81
	Summary	83
VI.	Quasilinearization Method	85
	Proposed Method	86
	Moving to the Surface	88
	Obtaining the Spline	89
	Jumping	90
	Algorithm	93
	Analysis of Population of Local Maxima	93
	Summary	97
VII.	Conclusions and Recommendations	99
	Conclusions	99
	Recommendations	101
	N-bit Simulator	102

Mathematical Routines	103
Avionics Routines	103
Modified Quasilinearization Method	103
Bibliography	105
Appendix A: User's Manual for the N-bit Simulator	110
Appendix B: N-bit Simulator Subroutines Source Listing	116
Appendix C: Mathematical Function Approximations	133
Vita	142

List of Figures

Figure		Page
1	N-bit Simulator Without Transmitted Errors	5
2	N-bit Simulator With Transmitted Errors	6
3	User Specifications (Options)	11
4	Nonlinear Relationships Representing Rounding and Truncation	15
5	Radix Effects	16
6	ONETRNC Examples	23
7	TWOTRNC Examples	25
8	ROUNDER Examples	26
9	Absolute Error of Truncated CDC Square Root	47
10	Relative Error of Truncated CDC Square Root	47
11	Absolute Error of Square Root Solution 2	50
12	Relative Error of Square Root Solution 2	50
13	Absolute Error of Square Root Solution 3	53
14	Relative Error of Square Root Solution 3	53
15	Absolute Error of Sine Approximation on $[0, \pi/2]$	59
16	Relative Error of Sine Approximation on $[0, \pi/2]$	59
17	Absolute Error of Sine Approximation on $[0, 2\pi]$	60
18	Relative Error of Sine Approximation on $[0, 2\pi]$	60
19	Absolute Error of Truncated CDC Sine on $[-2\pi, 2\pi]$	61
20	Relative Error of Truncated CDC Sine on $[-2\pi, 2\pi]$	61
21	Absolute Error of Cosine Approximation on $[0, 2\pi]$	64
22	Relative Error of Cosine Approximation on $[0, 2\pi]$	64
23	Bearing to Go	67
24	Code to be Compiled Without Being Preprocessed	70

Figure		Page
25	Code to be N-bit Simulated	71
26	COMPAR Flow Diagram	72
27	Absolute Error -vs- BTG Input Variable λ	75
28	Absolute Error -vs- BTG Input Variable λ_T	75
29	Absolute Error -vs- BTG Input Variable ϕ	76
30	Absolute Error -vs- BTG Input Variable ϕ_T	76
31	Absolute Error Using 5000 Numbers -vs- λ	78
32	Absolute Error Using 5000 Numbers -vs- λ_T	78
33	Absolute Error Using 5000 Numbers -vs- ϕ	79
34	Absolute Error Using 5000 Numbers -vs- ϕ_T	79
35	Absolute Error Bound for Sine (X) on $[0, 2\pi]$	87
36	Quasilinearization Jump	91
37	Quasilinearization Jump Flow Diagram	92
38	Empirical Distribution	95

List of Tables

Table		Page
1	Results of Kolmogorov-Smirnov Test	33
2	Results of Runs Test	35
3	Results of Serial Correlation Test	35
4	Minimax Sine Coefficients	58

Abstract

Errors due to finite wordlength are unavoidable when aircraft signal processing operations such as flight control, navigation, and fire control are implemented on a digital computer. To reduce these errors to tolerable levels, longer wordlengths can sometimes be employed. The effects of some of the errors, such as those due to arithmetic series truncation, machine roundoff, and quantization of system coefficients, can be lessened somewhat by appropriate numerical analysis techniques.

An n-bit simulator which runs on Control Data Corporation (CDC) 6600/CYBER 74 computer systems was modified and then used to evaluate the accuracy of a flight navigation routine coded in FORTRAN. The routines were executed without the simulator to obtain results used for benchmarking. The n-bit simulator was employed to simulate the numerical characteristics of the AN/AYK-15A digital processor. Error plots were constructed which show the maximum errors occurring within small plotting intervals plotted against each individual input value. These plots were used to aid visually in analyzing the error characteristics of the avionics routine as it would be implemented on the AN/AYK-15A.

A critical analysis of the error plots obtained showed that routines which are coded using single-precision floating-point arithmetic are prone to errors which exceed the error bounds specified for the routines. This occurs even though range reductions in the trigonometric function approximations are accomplished using extended precision.

I Introduction

When aircraft signal processing operations such as flight control, navigation, and fire control, are implemented on a digital computer, errors result which are due to finite computer wordlengths. Longer wordlengths must be employed sometimes to reduce these errors to tolerable levels. Appropriate numerical analysis and design techniques can reduce the effects of some of the errors, such as those due to arithmetic series truncation, machine roundoff, and quantization of system coefficients.

Two basic approaches might be considered when trying to analyze the numerical error characteristics of a programmed routine. First, consider the routine as given and determine the wordlength needed to give a desired accuracy. Second, considering specific hardware characteristics and the associated routine algorithm, attempt to implement a modified algorithm such that the resulting errors are minimized. In this investigation, the second approach was selected as the primary mode of analysis.

The Air Force Avionics Laboratory has written a development specification for the AN/AYK-15A computer processor (Ref 2). This processor will be used in the Digital Avionics Information System (DAIS) Integrated Test Bed and is a candidate for a follow-on flight test program in the F-16 aircraft. This specification establishes the performance, design, development, and test requirements for the Processor prime item. The instruction set specified is the MIL-STD-1750 airborne computer instruction set (Ref 49). MIL-STD-1750 defines the instruction set, mnemonics, and data format requirements for airborne computers.

Purpose

The purpose of this investigation is to develop tools and techniques which can be used to determine if a given computer can solve a given avionics signal processing problem within certain specified error and time tolerances. Specifically, the following goals were defined:

- develop a tool to simulate accurately the computational characteristics of any digital processor being studied
- produce common library routines which are optimal in the sense that they try to maximize both absolute and relative accuracy and at the same time minimize the number of instructions (especially multiplications) required
- demonstrate the effectiveness of the simulation tool mentioned above to analyze the error characteristics of a given class of algorithms and associated routines.

Approach

There are four basic requirements which must be satisfied in order to meet these goals. The first requirement is that the n-bit simulator must accurately simulate the numerical properties of processors which use the airborne computer instruction set defined in the MIL-STD-1750 document. The second requirement is that library routines for sine, cosine, arctangent, and square root be developed which produce results as accurately as possible. These routines must reflect the error characteristics they would have if they were implemented on the simulated computer. The third requirement is to demonstrate how the n-bit simulator can be used to perform an error analysis. This requirement was constrained to forward error analysis. The fourth requirement is to indicate how the techniques applied to one program representing an

algorithm can be applied to a larger class of programs. If the techniques demonstrated can be applied to a larger class of programs, then programs from this larger class can be analyzed to determine the compatibility of the programs with the computer being simulated.

Several procedures or approaches are available for assessing the quality of floating-point mathematical software. Typical analysis schemes can be classified as follows:

- 1) error-bounding schemes (Refs 9; 31; 50; and 53),
- 2) forward error analysis (Refs 10; 25; 29; 52; and 62),
- 3) backward error analysis (Refs 48; 59; 62; 65; and 66),
- 4) multiple-precision arithmetic (Refs 5; 17; 35; and 68),
- 5) perturbation analysis (Ref 63), and
- 6) significance arithmetic (Refs 3 and 6).

Each of these six approaches was considered, with forward error analysis being chosen. Forward error analysis requires that computed results be compared to higher precision reference values. This requirement was met by using the Control Data Corporation (CDC) CYBER 74 computer with its 60-bit wordlength to compute the higher-precision results. These results are then available to be compared to results which would be obtained using the computer on which the routines would normally be executed. Thus, the necessary tools and techniques for conducting a forward error analysis can be developed for use with the CDC CYBER 74 computer. Since forward error analyses reveal quantization, roundoff, and truncation errors as they actually occur (as opposed to just giving upper limits), this method was selected over the other error analysis schemes.

There are three primary sources of errors in any numerical result: transmitted errors, analytic truncation errors, and generated errors

(Ref 11:125). Transmitted errors are those which occur in the original input data. Analytic errors occur when finite processes are substituted for essentially infinite processes in the mathematical algorithm. The third source of errors, generated errors, represent the errors actually generated within the computer program if exact input data is entered. Generated errors reflect the method of rounding or truncation utilized and other design characteristics such as the number system base, or machine radix. For the purposes of this investigation, when a program is being evaluated solely for its accuracy, the transmitted errors are not considered, since they would normally occur during the conversion of analog signals to digital signals. A simplified diagram depicting the method utilized by the n-bit simulator to perform a forward error analysis is shown in Fig. 1. For simulations where the transmitted errors are included, the n-bit simulator will be employed as indicated in Fig. 2. In both diagrams, the area within the dotted lines represents the phase of analysis using n-bit accuracy. A general discussion of the philosophical questions concerning the isolation of transmitted errors is presented by Kuki and Ascoly (Refs 41 and 42).

Tools and Techniques

Several software tools exist to aid the analyst in performing a forward error analysis. Two of these are an n-bit wordlength simulator (Ref 37) and a floating-point simulator (Ref 25). The n-bit simulator was selected as the primary tool for several reasons. First, with a few modifications it should be able to simulate the numerical properties of any processor following the MIL-STD-1750 specification; second, it is well documented; and third, it has been tested on a CDC CYBER 74 computer system, thereby eliminating the need to use multiple-precision

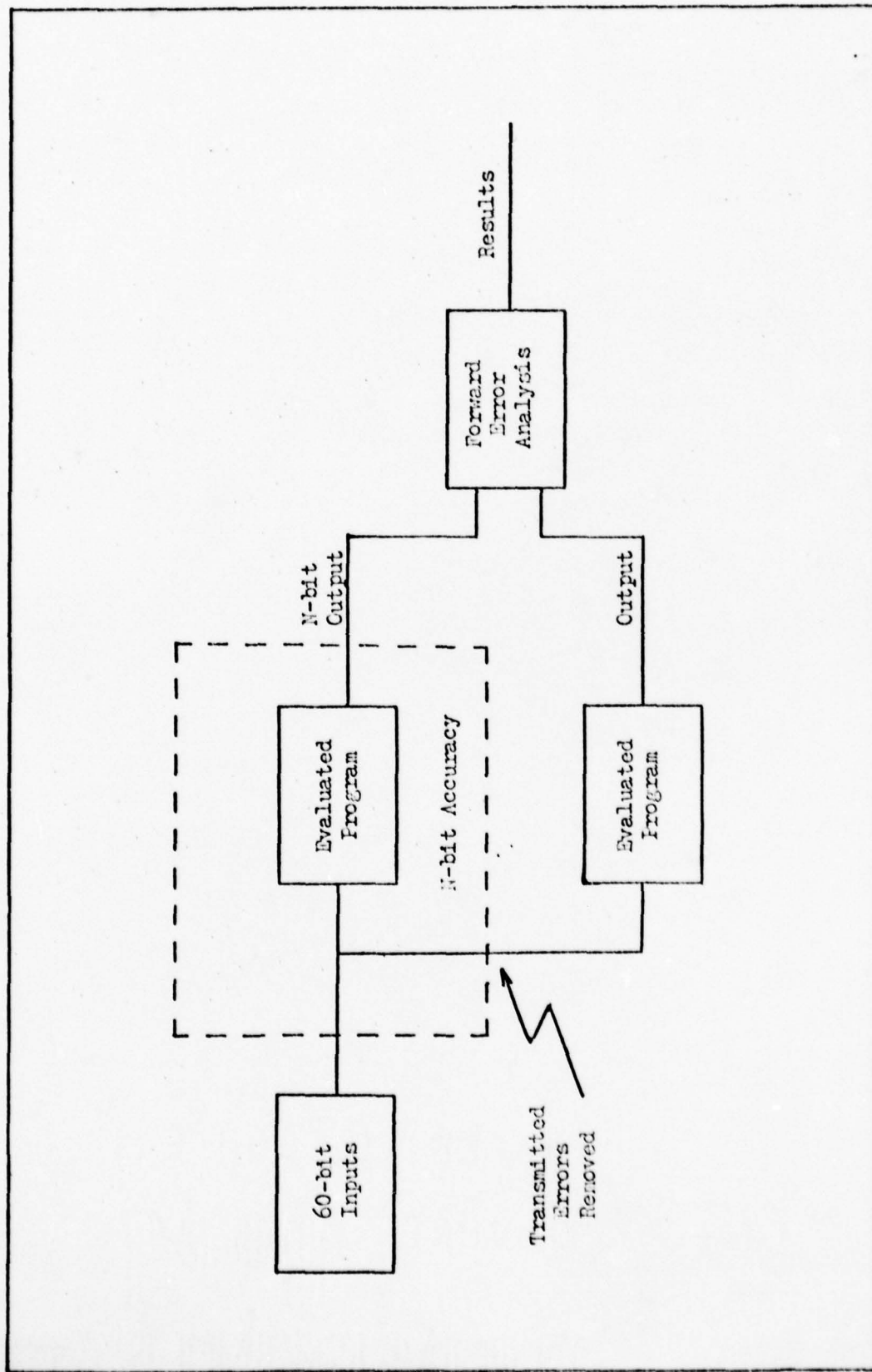


Fig. 1. N-bit Simulator Application Without Transmitted Errors

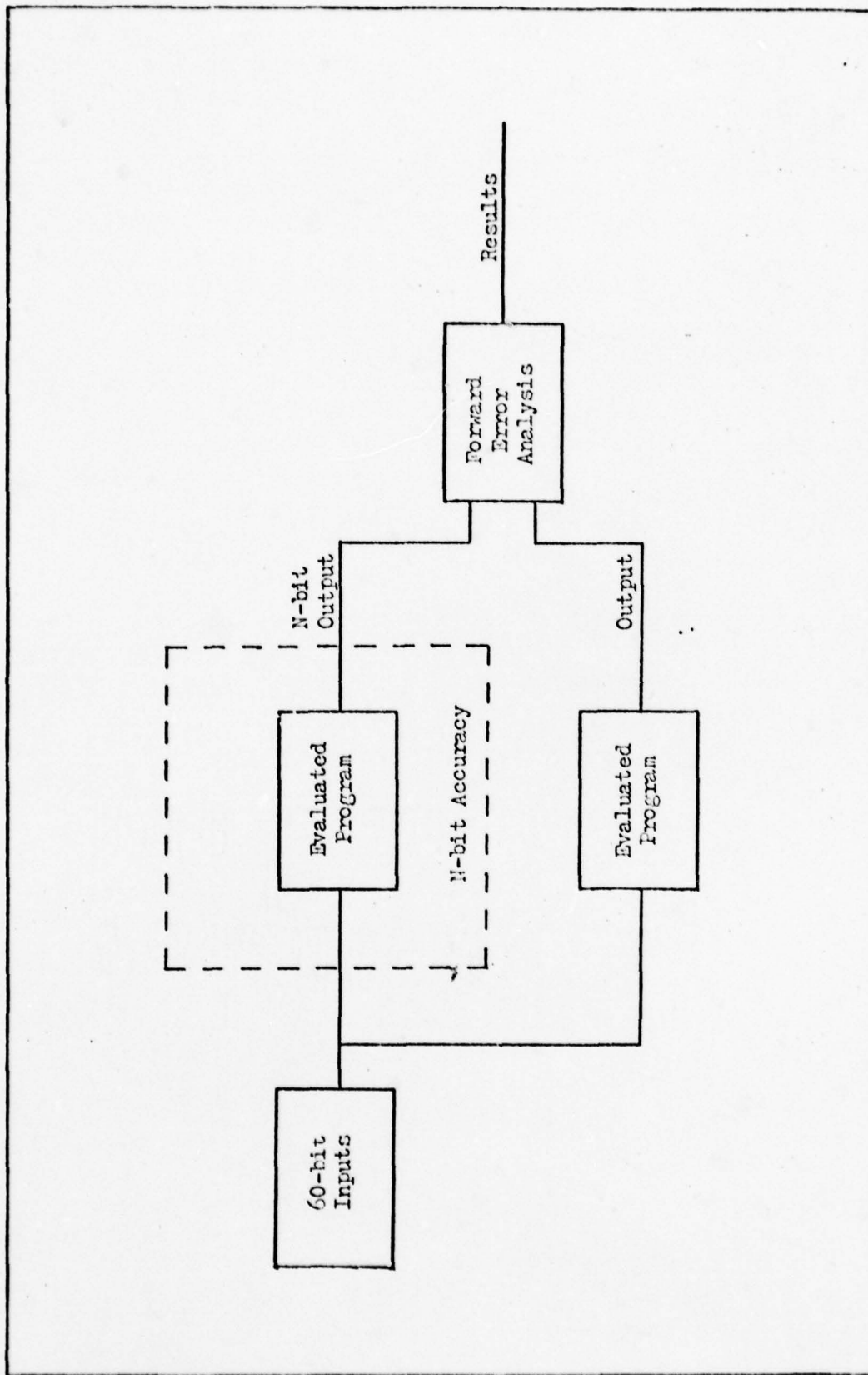


FIG. 2. N-bit Simulator Application With Transmitted Errors

packages in order to obtain benchmark data. This also eliminated the need to use two computers, since the higher-precision results and the results of the simulated computer can be made available at the same time for immediate comparison using the accuracy of the CDC CYBER 74 computer. The n-bit simulator gives the user the option of selecting different wordlengths within any program, thereby making it a valuable tool for perturbation analysis, since the processor wordlength is also considered to be a candidate for perturbations. It is assumed that the reader is familiar with the work by Klein (Ref 37).

Assumptions/Constraints

Several assumptions were made during the course of conducting this investigation. Five of these assumptions are the same as the first five assumptions by Klein (Ref 37:6-8). These five assumptions cover various aspects of the n-bit simulator such as its capabilities and the environment in which it is executed. Klein's sixth assumption has been changed due to the modifications made to the n-bit simulator. The n-bit simulator is now able to simulate computers using one's complement, two's complement, or sign-magnitude arithmetic. Also, binary, quaternary, octal, or hexadecimal numerical data representations can now be simulated. There is still no provision for simulating decimal representations. The first part of Klein's sixth assumption, that floating-point mantissas will be normalized, remains the same. Five other assumptions that were made are:

- the CDC CYBER 74 mantissa length of 48 bits provides sufficient accuracy (error less than one tenth of one percent of that produced by the simulated computer) for benchmarking purposes in a forward error analysis,

- fixed-point numbers would be used primarily for addressing, sub-scripting, and logical values, while floating-point numbers would be used in performing the numerical computations (Ref 2),
- execution time is a "precious resource" on avionics computers, meaning that flight routines must execute as efficiently as possible while still maintaining the required accuracy (Ref 57),
- coding an algorithm in FORTRAN for the purpose of executing it on the n-bit simulator will not significantly alter its error characteristics from those it would have if implemented on an avionics computer, since the n-bit simulator modifies all intermediate results as well as the final results, and
- avionics algorithms found in Ref 57 are representative of general avionics algorithms which might be programmed for use in different aircraft and are therefore sufficient to be employed in developing a method for evaluating a general class of algorithms. Since these algorithms are for the F-16 aircraft, they can be used to test the error characteristics of the AN/AYK-15A avionics processor which is a candidate for use in F-16 aircraft.

Chapter Synopsis

The second chapter of this investigation contains a summary of the modifications made to the n-bit simulator to allow it to simulate accurately processors which use the airborne computer instruction set defined by MIL-STD-1750, and in particular, the AN/AYK-15A processor. The third chapter contains a discussion of Monte Carlo techniques which were applied when conducting a forward error analysis. Topics discussed include generating and testing pseudo-random numbers and obtaining criteria to use to decide when to stop testing. The fourth chapter contains a

discussion of the procedures used to develop sine, cosine, and square root function approximations. There was no approximation developed for the arctangent function. Many different approximations and methods were tested, with only the "best" being discussed. The fifth chapter contains a detailed discussion of one flight routine (representing a portion of an algorithm) and the associated error analysis conducted. This chapter presents a method for using the n-bit simulator in forward error analysis studies on this one flight routine. The techniques discussed are shown to be applicable for testing other routines as well. The sixth chapter contains a discussion of the quasilinearization method as it might be applied in a forward error analysis. The last chapter contains the conclusions drawn from the error analyses and associated results and discusses recommendations for future research.

II N-bit Simulator

The n-bit simulator (Ref 37) consists of two major components. The first, the preprocessor, reads as input a FORTRAN source program representing an algorithm, translates it into another FORTRAN source program with all the arithmetic operations replaced by subroutine calls, and writes it to an output file. The second part is a collection of subroutines the modified FORTRAN source program calls when executing. These subroutines perform the arithmetic operation they replaced using the wordlength of the host computer and then they modify the result to reflect the properties of the simulated computer processor wordlength. Some of the important characteristics of the n-bit simulator are its capabilities to handle both fixed-point (limited) and floating-point arithmetic, its capability to perform either rounding or truncation, and its documentation and overall design which allowed it to be easily modified. In this chapter are discussed the original n-bit simulator, its capabilities, and a series of modifications made to the preprocessor and selected subroutines. Options were added to allow the user to specify the machine radix, the number of guard bits used, and the type of arithmetic performed (one's complement or two's complement). The user also now has the capability of specifying the fixed-point wordlength separately from the floating-point wordlength. A detailed discussion of the original n-bit simulator is presented by Klein (Ref 37). An updated user's manual is shown in Appendix A.

Preprocessor

The preprocessor changes all arithmetic assignment statements into a series of one or more subroutine calls. Each arithmetic operator (+, -, *, /, **, and =) is replaced by one subroutine call, thereby

allowing both intermediate and final results to reflect accurately the characteristics of the simulated computer. The only change made to the preprocessor was the addition of one more parameter in the subroutine calls. This extra parameter, which is the name of an array which holds the overflow and underflow limits for floating point values, was added to allow faster checking for floating-point overflow and underflow. The values in the original simulator were stored in an integer array and the SHIFT function was used to prevent conversion of the values to fixed-point format.

SETNBIT Subroutine

SETNBIT is the name of a subroutine which is called as the first executable statement of the FORTRAN program and wherever else required. This subroutine takes the input parameters which are the user options and converts them into values the numerical routines need. The ten user options associated with any call to the SETNBIT routine are shown in Fig. 3.

User specifications:	
1) NBITS	Floating point wordlength
2) MANTSA	Mantissa length (excluding sign)
3) IGUARD	Number of guard digits
4) IEXPNT	Exponent length (including sign)
5) IPTPOS	Binary point position
6) IRNDTR	Round or truncate
7) IONTWO	One's complement or two's complement arithmetic
8) ITYPE	Machine radix
9) MESSGS	Message suppression
10) IFIXD	Fixed point wordlength

Fig. 3. User Specifications (Options)

Each entry shows the order the option occurs in the parameter list, the variable name, and its meaning. The variable names will be used in

equations later in this chapter. The output values are stored in two arrays, the first (KEY) holding fixed-point values and the second (TKEY) holding floating-point values. These arrays are the last two parameters for SETNBIT. The ten user options will be explained first followed by the way the two arrays are filled.

Option One. The first option is used to indicate the total number of bits in a floating-point word. As an example of this, the AN/AYK-15A digital processor has a machine wordlength of 16 bits, with single precision fixed-point words being represented by 16 bits, or one machine word, and single-precision floating-point words being represented by 32 bits, or two machine words. When simulating with single precision floating-point words, the value of option one would be set to 32, or the number of bits in the floating-point word. Since some smaller computers use different numbers of machine words to represent fixed-point and floating-point words (Ref 49), just as the AN/AYK-15A does, option ten was added to allow the user to specify a fixed-point wordlength different from the floating-point wordlength specified in option one.

Option Two. The second option allows the user to specify the number of bits in the mantissa of the floating-point word, not counting the sign of the mantissa. This is the mantissa length for all final variable assignments.

Option Three. The third option, which replaces the one in the original simulator which was used to indicate single, double, or triple precision computation, is used to indicate the number of guard bits employed by the simulated computer. The uses of guard digits as defined by Kuki and Cody (Ref 43:224) are as follows. For addition, guard

digits are primarily used to retain shifted digits of the operand with the smaller exponent at the time of the exponent alignment of the operands. Guard digits also participate in the right shift of the intermediate sum in case of a carry. Using only one guard digit, the relative accuracy of the sum of two exactly represented numbers may be protected to within machine precision. The representation of the product of two N-digit significands requires 2N digits. If, of these 2N digits, N+K high order digits are actually developed before the postnormalization of the result, then K guard digits have been used for multiplication. Since the multiplication of two normalized operands requires postnormalization of at most one digit, there will be either K or K-1 guard digits available for possible rounding after the postnormalization. If no guard digits are used, a number could be changed by multiplying it by 1.0 (Ref 43:224). It is still possible to indicate single, double, or triple precision computations by setting this option to the required number of extra bits. As an example of how this option is used, if MANTSA is assigned the value 23 and IGUARD the value 2, the multiplication

$$\text{TEMP} = T1 * T2 \quad (1)$$

would be performed using 25-bit mantissas and the result would be stored in the variable location TEMP using a 23-bit mantissa. For more information on the effects of guard digits see Kuck, Parter, and Sameh (Ref 39), Kuki and Cody (Ref 43), and Cody (Ref 12).

Option Four. Option four is the same as one of the options in the original simulator and is used to show the exponent length for floating-point words. The sign of the exponent is included in the length of the exponent, so the condition

$$\text{NBITS} = \text{MANTSA} + \text{IEXPNT} + 1$$

(2)

must hold true.

Option Five. The fifth option is the same as one in the original simulator and allows the user to specify whether the binary point is to the left of the mantissa (making the mantissa a fraction) or to the right of the mantissa (making the mantissa an integer).

Option Six. This option is also the same as one in the original simulator and is used to specify whether rounding or truncation will be performed.

Option Seven. This option was added to allow the user to specify whether the machine being simulated words in sign plus magnitude, sign plus one's complement, or sign plus two's complement floating-point arithmetic. When rounding is being performed, the only differences are in the overflow and underflow values. When truncation is being performed, however, one other rather subtle difference appears, that being that sign plus two's complement negative numbers are truncated away from zero while sign plus magnitude and sign plus one's complement negative numbers are truncated toward zero. This concept, shown graphically in Fig. 4 for fixed-point numbers, is explained in detail for both fixed-point and floating-point numbers by Oppenheim and Weinstein (Ref 55:406-413). In Fig. 4, x represents the number before truncation or rounding, $Q(x)$ represents the number after truncation or rounding, and the variable b represents the wordlength for a fixed-point number. Rounding and truncation are performed by the special subroutines ONETRNC, TWOTRNC, and ROUNDER which are explained in detail later in this chapter. For more information on the effects of rounding and truncation, see Kuck, Parker, and Sameh (Ref 39), Tsao (Ref 61), Kuki

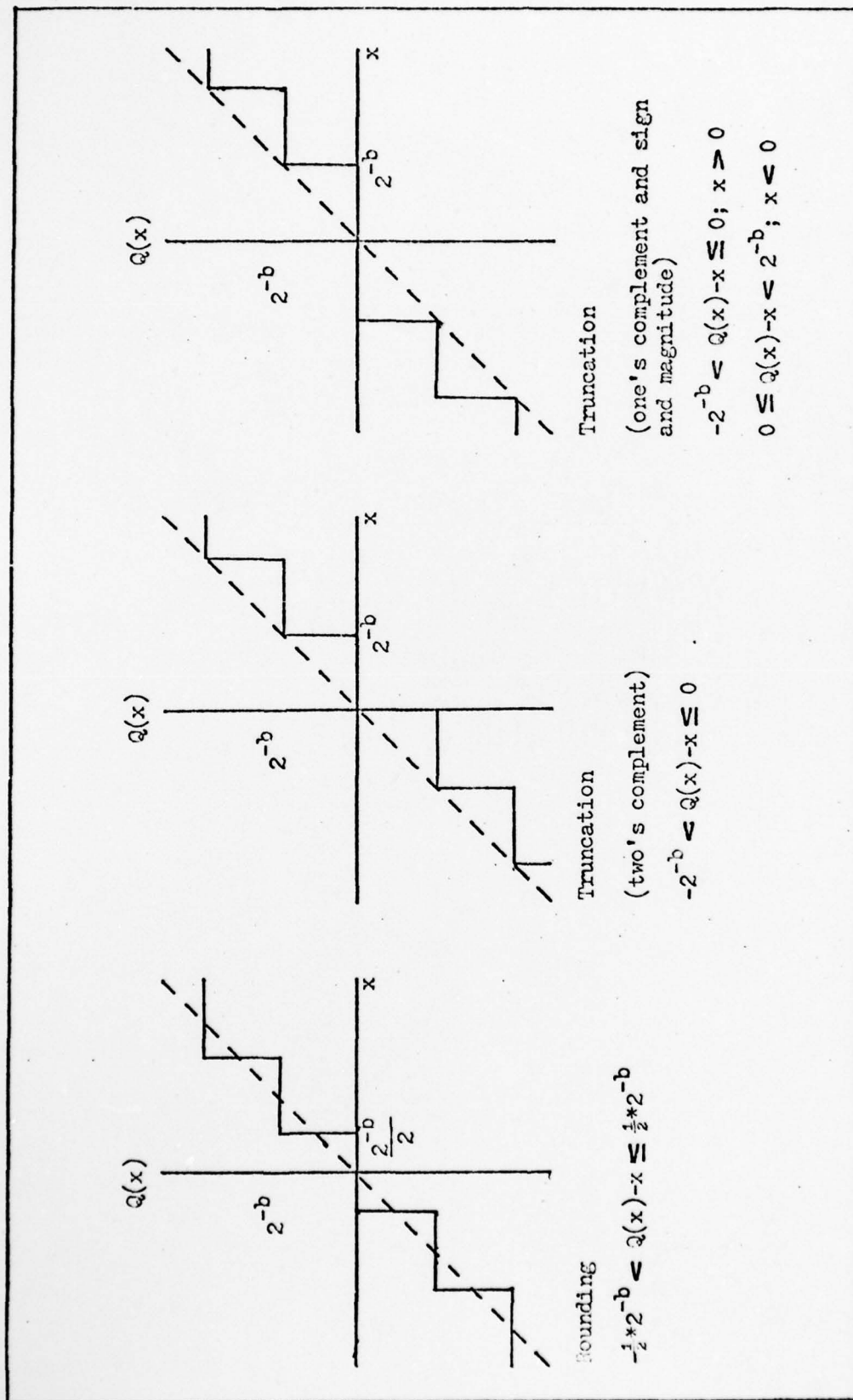


Fig. 4. Nonlinear Relationships Representing Rounding and Truncation (Ref 55:412)

and Cody (Ref 43), Cody (Ref 12), and Kaneka and Liu (Ref 36).

Option Eight. Option eight was added to allow the user to specify the machine base, or radix, of the computer being simulated, provided the radix is a multiple of two. Binary, octal, and hexadecimal machines are examples of computers having a machine base which is a multiple of two. The machine base defines the actual value of the exponent and affects not only the overflow and underflow limits, but also the way the mantissa is normalized. In the following examples, the mantissa sign bit is first, the mantissa is second, and the exponent is last, since this is the way computers following the MIL-STD-1750 document represent floating-point numbers. Truncation is assumed, and the four bit patterns in Fig. 5 represent how the result of dividing 127 by 64, or the number 1.984375, would be represented on binary, quaternary, octal, and hexadecimal machines respectively.

Bit pattern	Actual value	Radix
0 1111111 0001	1.984375	2
0 0111111 0001	1.96875	4
0 0011111 0001	1.9375	8
0 0001111 0001	1.875	16

Fig. 5. Radix Effects

In each case, the wordlength is 12 bits, with 1 bit for the sign, 7 bits for the mantissa, and 4 bits for the exponent. More information on the effects that the machine radix has on numerical accuracy can be found in Kuki and Cody (Ref 45), Kuck, Parker, and Sameh (Ref 39), Goldberg (Ref 27), and Cody (Ref 12).

Option Nine. The ninth option is the same as one in the original

simulator and is used to control the printing or suppression of overflow and underflow messages. If, during any part of the simulation, overflow occurs on the CDC CYBER 74 and the value is used again, the program will terminate regardless of the value of this flag.

Option Ten. This last option was added to allow the user to specify the fixed-point wordlength so that correct overflow bounds can be established for both fixed-point and floating-point computations. This allows programs run on computers such as the AN/AYK-15A, where the fixed-point wordlength is different from the floating-point wordlength, to be simulated accurately without having to call the SETNBIT subroutine when changing between fixed-point and floating-point computations.

KEY. The array KEY holds fixed-point values which are used by the numerical subroutines. The following paragraphs describe how the elements of the KEY array are filled from the values specified as user options.

KEY(1). KEY(1) holds the largest fixed-point positive number which can be represented on the machine being simulated. Given the IFIXD input in option ten, KEY(1) may be computed by

$$\text{KEY}(1) = (2^{**}(\text{IFIXD}-1))-1 \quad (3)$$

KEY(2). KEY(2) holds the largest (magnitude) negative value of the simulated computer and is used in checking for negative fixed-point overflow. For sign plus magnitude and sign plus one's complement machines, KEY(2) may be computed by

$$\text{KEY}(2) = -\text{KEY}(1) \quad (4)$$

but for sign plus two's complement machines it should be computed by

$$\text{KEY}(2) = -(2^{**}(\text{IFIXD}-1)) \quad (5)$$

KEY(3). KEY(3) holds user option six, or IRNDTR, specifying

rounding or truncation. The use of this field is discussed further later in this chapter in the section entitled Special Subroutines.

KEY(4). KEY(4) holds user option seven, or IONTWO, specifying one's complement or two's complement arithmetic. The use of this field is also discussed later in this chapter in the section entitled Special Subroutines.

KEY(5). KEY(5) holds user option nine, or MESSGS, specifying print or print suppression.

KEY(6). KEY(6) holds the number of mantissa bits to save on all final assignments and is filled with the value of MANTSA, or user option two.

KEY(7). KEY(7) holds the number of extra mantissa bits to save during intermediate operations and is filled with the value of IGUARD, or user option three.

KEY(8). KEY(8) holds user option eight, or ITYPE, specifying the radix of the machine being simulated. The value actually stored is the exponent of two which would give the desired radix (the value 3 represents octal, since $2^{*3}=8$).

TKEY. The array named TKEY holds four values used for overflow and underflow checking. While each element of the array can be represented by a mathematical expression, the actual values are computed using shift, mask, and addition or subtraction operations. In some instances, an operand being added or subtracted is unnormalized, with the answer being normalized by the CDC CYBER 74 computer. If the physical limits of the CDC computer would be exceeded in trying to represent the overflow or underflow values of the computer being simulated, the CDC limits are used instead. The documented code to implement the algorithms to

fill the TKEY array is shown in Appendix B. Those operands used in additions or subtractions which are unnormalized are annotated as such. In the equations which follow, the variables are the same as those shown in Fig. 3.

TKEY(1). The first element of array TKEY holds the largest positive floating-point value that can be expressed on the computer being simulated. If, during any operation, a floating-point result exceeds this value, positive overflow will be signaled and the result will be replaced by TKEY(1). The value stored in TKEY(1) depends upon the location of the binary point specified by IPTPOS, or user option 5. If the binary point is on the left, TKEY(1) may be computed using

$$\text{TKEY}(1) = (2^{**\text{MANTSA}-1}) * (2^{**(\text{ITYPE} * ((2^{**(\text{IEXPNT}-1))}-1)} - \text{MANTSA})) \quad (6)$$

If the binary point is on the right, TKEY(1) may be computed using

$$\text{TKEY}(1) = (2^{**\text{MANTSA}-1}) * (2^{**(\text{ITYPE} * ((2^{**(\text{IEXPNT}-1))}-1)))} \quad (7)$$

TKEY(2). The second element of TKEY holds the largest (magnitude) negative floating-point value representable by the computer being simulated. If any result is less than TKEY(2), negative overflow is signaled and the result is replaced by TKEY(2). The value stored in TKEY(2) depends upon both the location of the binary point specified by IPTPOS and the type of arithmetic specified by IONTWO, or user option 7. If the binary point is on the left and sign plus one's complement arithmetic is specified, TKEY(2) may be computed using

$$\text{TKEY}(2) = -(2^{**\text{MANTSA}-1}) * (2^{**(\text{ITYPE} * ((2^{**(\text{IEXPNT}-1))}-1)} - \text{MANTSA})) \quad (8)$$

If the binary point is on the left and sign plus two's complement arithmetic is specified, TKEY(2) should be computed using

$$\text{TKEY}(2) = -(2^{**}(\text{ITYPE}*((2^{**}(\text{IEXPNT}-1))-1))) \quad (9)$$

For the cases where the binary point has been specified to be on the right,

$$\text{TKEY}(2) = -(2^{**}\text{MANTSA}-1)*(2^{**}(\text{ITYPE}*((2^{**}(\text{IEXPNT}-1))-1))) \quad (10)$$

is used for the sign plus one's complement case and

$$\text{TKEY}(2) = -(2^{**}(\text{ITYPE}*((2^{**}(\text{IEXPNT}-1))-1)+\text{MANTSA})) \quad (11)$$

is used for the sign plus two's complement case.

TKEY(3). The third element of TKEY holds the smallest positive floating-point number which is greater than zero and still representable on the computer being simulated. If any result is greater than zero and less than TKEY(3), positive underflow is signaled and the result is replaced by zero. For the cases where the binary point is specified to be on the left,

$$\text{TKEY}(3) = 2^{**}(-(\text{ITYPE}*((2^{**}(\text{IEXPNT}-1))-1)+\text{ITYPE})) \quad (12)$$

is used when sign plus one's complement arithmetic is specified and

$$\text{TKEY}(3) = 2^{**}(-(\text{ITYPE}*(2^{**}(\text{IEXPNT}-1))+\text{ITYPE})) \quad (13)$$

is used when sign plus two's complement arithmetic is specified. If the binary point is specified to be on the right, the value for TKEY(3) is computed using

$$\text{TKEY}(3) = 2^{**}(-(\text{ITYPE}*((2^{**}(\text{IEXPNT}-1))-1)+\text{ITYPE}-\text{MANTSA})) \quad (14)$$

for the sign plus one's complement case and using

$$\text{TKEY}(3) = 2^{**}(-(\text{ITYPE}*(2^{**}(\text{IEXPNT}-1))+\text{ITYPE}-\text{MANTSA})) \quad (15)$$

for the sign plus two's complement case.

TKEY(4). The fourth element of TKEY holds the smallest (magnitude) negative floating-point value which is less than zero and still representable on the computer being simulated. If any result is greater than TKEY(4) and less than zero, negative underflow is signaled

and the result is replaced by zero. If the binary point has been specified to be on the left,

$$TKEY(4) = -(2^{**}(-(ITYPE*((2^{**}(IEXPNT-1))-1)+ITYPE))) \quad (16)$$

should be used if sign plus one's complement arithmetic has been specified and

$$TKEY(4) = -((2^{**}(-ITYPE))+(2^{**}(-MANTSA)))*(2^{**}(-(ITYPE*(2^{**}(IEXPNT-1))))) \quad (17)$$

should be used if sign plus two's complement arithmetic has been specified. For the cases where the binary point has been specified to be on the right,

$$TKEY(4) = -(2^{**}(-(ITYPE*((2^{**}(IEXPNT-1))-1)+ITYPE-MANTSA))) \quad (18)$$

is used for the sign plus one's complement case and

$$TKEY(4) = -((2^{**}(-ITYPE))+(2^{**}(-MANTSA)))*(2^{**}(-(ITYPE*(2^{**}(IEXPNT-1))-MANTSA))) \quad (19)$$

is used for the sign plus two's complement case.

Function Subroutines

The function subroutines are called to perform all additions, subtractions, multiplications, divisions, exponentiations, and assignments. The subroutines, together with the operation they perform and the types of operands they have, are shown on page 47 of the thesis by Klein (Ref 37). Although several changes were made to these subroutines, their basic concept remains the same. The first change is that they all now have one extra parameter, the array TKEY. The subroutines which handle only fixed-point operands do not use this parameter, but more extensive coding changes would have been required in the preprocessor to differentiate between the subroutines which have at least one floating-point operand and those which do not. The second change was

necessitated when the user option to handle sign plus two's complement arithmetic was added. Separate checks are made for positive and negative overflow in all the subroutines, and for those subroutines having at least one floating-point operand, there are also separate checks for positive and negative underflow. The code for printing the overflow and underflow messages was put into the function subroutines, eliminating the small print subroutine that was part of the original simulator. The third change is that all the code to accomplish the rounding and truncation for floating-point operations has been put into three special subroutines.

Special Subroutines

The three special subroutines, called `ROUNDER`, `ONETRNC`, and `TWOTRNC`, perform rounding, sign plus one's complement truncation, and sign plus two's complement truncation on floating-point results. The documented code for these subroutines is shown in Appendix B. The option which allows the user to specify the radix of the machine (`ITYPE`) increased the complexity of the code considerably. The exponent must be examined before determining the number of bits to save in the mantissa, since on a machine with a radix greater than two, the mantissa might be shifted one or more places to the right. To simulate this happening, the number of places the mantissa would be shifted must be computed and then subtracted from the number of mantissa bits to save. For intermediate results, the number of guard digits is added to give the total number of mantissa bits to save.

ONETRNC. The subroutine `ONETRNC` performs sign plus one's complement truncation just as it was performed in the original simulator subroutine, except it has added code to handle the machine radix option

and the guard bits option. A step-by-step example of how ONETRNC works is shown below in Fig. 6. This example shows how the result of 1023 divided by 512 would be truncated for an octal machine with a word-length of 12 bits, 7 of which are mantissa. There is 1 guard bit and the number being truncated is an intermediate result.

a) 0 0011111 1 0001
b) 17207774000000000000
c) 00000000000000001720
d) 0000000000000000172077
e) 17207700000000000000

Fig. 6. ONETRNC Examples

Line (a), which is in binary, shows what the result would look like on the simulated machine. Line (b) shows what the answer looks like on the CDC CYBER 74 computer before being manipulated by ONETRNC. Line (c) shows the first 12 bits of the CDC CYBER 74 word after they have been shifted right 48 places, putting the exponent into fixed-point position. The word has been filled with the mantissa sign during the shift, and following the shift, if the result is negative, it is complemented. The value at this point represents the biased exponent of the CDC CYBER 74 word. If the sign of the exponent and mantissa are the same, then the exponent is subtracted from 2056, otherwise the exponent is subtracted from 2055. This subtraction allows the FORTRAN MOD function to compute the number of extra mantissa shifts required to compensate for the machine radix. The MOD function computes the remainder of this difference divided by the base two logarithm of the machine radix, or ITYPE, which is stored in KEY (8). In this example, the exponent (976) was

subtracted from 2055, giving 1079, and ITYPE is 3, so the number of extra bits lost is 2, which is also the number of leading zeros in the mantissa in line (a). The number of bits to truncate for intermediate results is found by subtracting the number of mantissa bits and guard bits to save from 48, then adding the number to lose due to the machine radix. For this example, the final number of bits to lose is $48 - (7 + 1) + 2$, or 42 bits, leaving 6 significant bits in the CDC CYBER 74 mantissa. The CDC CYBER 74 word is then shifted 42 places, first to the right with a sign extension fill (line d) and then to the left with circular fill (line e). The result has the same numerical value as line (a).

TWOTRNC. TWOTRNC works just like ONETRNC except that for negative numbers, TWOTRNC truncates them away from zero instead of toward zero. This is done by adding to the negative number a negative number which is composed of the same exponent value and an unnormalized mantissa which may be computed by treating the mantissa like a fractional value and subtracting the smallest non-zero simulated mantissa from the smallest nonzero unnormalized CDC CYBER 74 mantissa. The example in Fig. 7 shows the step-by-step two's complement truncation of the result of dividing -1023 by 512 on a binary machine with a 12-bit wordlength. The mantissa length is 7 bits, there are no guard bits, and the binary point is on the left. Line (a), which is in binary, shows what the result should look like on the simulated computer. Line (b) shows what the result looks like on the CDC computer before being manipulated by TWOTRNC. Line (c) shows the unnormalized number which is to be added to the result. Line (d) shows the result of this addition. The next step is to perform the shifting just as in ONETRNC. Line (e) shows the result after being shifted right 41 places, and line (f) shows the final

a)	1 0111111 0010
b)	6057000377777777777
c)	6057774000000000000
d)	6056376200000000000
e)	7777777777777777413477
f)	6056377777777777777

Fig. 7. TWOTRNC Examples

result after being shifted left 41 places. The values in line (f) and line (a) are the same.

ROUNDER. The subroutine ROUNDER has been changed from the one in the original simulator in that now it handles all rounding which needs to be done. Rounding is accomplished in the same manner for both sign plus one's complement and sign plus two's complement arithmetic, so no distinction is made between the two. ROUNDER works the same as ONETRNC and TWOTRNC in the way that the machine radix and guard digits are handled. It also performs a sign plus one's complement truncation as the last step, just as the others do. Before the final shifts are performed, however, ROUNDER rounds by adding to the operand an unnormalized number which has the same sign and exponent as the operand. The mantissa of this number is computed by putting a 1 one place to the right of the computed end of the simulated mantissa and zeros elsewhere. For a negative number, this mantissa is then complemented. Fig. 8 shows an example of how the result of dividing 1023 by 512 is rounded to a 7-bit mantissa. No guard digits are used, and the machine is hexadecimal. Line (a) shows what the result would look like on the simulated computer, and line (b) shows what the result looks like on the CDC CYBER 74 before

a)	0 0010000 0001
b)	17207774000000000000
c)	17200200000000000000
d)	17214076000000000000
e)	00000000000000036430
f)	17214000000000000000

Fig. 8. ROUNDER Examples

being manipulated. Line (c) shows the unnormalized number which is added to the operand before truncation occurs. In this case, only 4 mantissa bits will be saved, since 3 are lost because of the machine radix. Line (d) shows the result of the addition, with the CDC handling the special case where the exponent has been changed. Line (e) shows the result of the left shift of 44 places, and line (f) shows the final result, which is the same as line (a).

Summary

Several changes were made to the original n-bit simulator to enable it to simulate more accurately various computer wordlengths. The pre-processor was modified so that it builds an extra argument into the function subroutine calls that it places in the modified FORTRAN source program. This extra argument, TKEY, is the name of a floating-point array which holds the floating-point overflow and underflow limits of the simulated computer. The subroutine SETNBIT was changed to give the user more flexibility in simulating different computers. Options were added to allow the user to differentiate between sign plus one's complement and sign plus two's complement machines, to specify the radix of the machine being simulated, to specify different floating-point and

fixed-point wordlengths, and to specify the presence of guard bits. The insertion of the guard-bit option eliminated the need for the option in the original simulator which allowed the user to specify whether computations were performed in single, double, or triple precision. The function subroutines were changed so that all overflow and underflow checks and messages are handled in the subroutines, and all rounding and truncation is done by three special routines called ONETRNC, TWOTRNC, and ROUNDER. Creating these special routines increased the execution time of the simulator, but at the same time it allowed for increased system reliability and maintainability.

III Monte Carlo Testing

When evaluating numerical software, one finds a direct relationship between the effort expended in testing and the confidence in the correctness of the software. One way to assure correctness is to test all possible combinations of inputs. Since this is rarely practical, a common practice is to use a limited set of input data for testing. In this chapter are discussed techniques for testing pseudo-random number generators and Monte Carlo techniques for testing numerical software. Also mentioned are testing methods proposed by Cody (Ref 10) and others to increase the probability that numerical accuracy problems in a given numerical routine will be discovered.

Generating Pseudo-Random Samples

Only binary computers which perform two's complement arithmetic are considered in this chapter, and it is assumed that floating-point mantissas are normalized. Floating-point numbers are represented with mantissas of length m bits and exponents of length e bits. Since the mantissas are normalized, there are only 2^{m-1} unique mantissa values. There is one extra mantissa representation which is used for the value 0 which is not normalized, since the mantissa is all zeros. Not counting the right endpoint of the interval $[0.0, 1.0]$, there are K unique numbers expressible where K may be computed by

$$K = (2^{m-1}) * (2^{e-1} + 1) + 1 \quad (20)$$

For the AN/AYK-15A computer, the mantissa length is 23 bits and the exponent length is 8 bits. Therefore, from (20), K is

$$K = (2^{23-1}) * (2^{8-1} + 1) + 1 = 541,065,217 \quad (21)$$

Thus, when testing even a simple routine such as a sine routine for

$\sin(\pi X/2)$ over the interval $[0.0, 1.0]$, it becomes impractical to test using all possible inputs. Therefore, Monte Carlo techniques are often employed.

When taking samples of input data, Cody (Ref 10) states that the input interval can be divided into subintervals, with a collection of random bit patterns tested in each subinterval. Not all numbers in his subintervals necessarily have the same exponent. For any of his subintervals containing numbers for which the error varies substantially, the subintervals are broken down further with more random bit patterns being chosen from each new subinterval.

If a subinterval is considered to contain all representable numbers with the same exponent, then the numbers are evenly spaced over the subinterval. This does not mean that one can take uniform random samples from the interval $[0.0, 1.0]$ and expect to get an unbiased random sampling of all possible values occurring in that interval. If so, half of the samples would be expected to lie in the interval $[0.5, 1.0]$. These numbers have an exponent of 0 and are uniformly-distributed over the subinterval $[0.5, 1.0]$. For the AN/AYK-15A computer, there are 129 ($2^{e-1} + 1$ where $e=8$) different exponents which, when combined with mantissas, produce numbers in the interval $[0.0, 1.0]$. Therefore, less than one percent of all representable numbers in the interval $[0.0, 1.0]$ actually lie in the subinterval $[0.5, 1.0]$.

Cody recommends using random bit patterns, and these can be obtained easily using the CDC pseudo-random number generator. The pseudo-random number generator returns values which are uniformly-distributed over the interval $[0.0, 1.0]$. Those numbers less than 0.5 can be modified by

with the resulting numbers being uniformly-distributed over the subinterval $[0.5, 1.0]$. The exponent can then be modified to provide a uniformly-distributed pseudo-random sample from any of the 129 subintervals making up the interval $[0.0, 1.0]$. Since all subintervals contain uniformly-distributed numbers, sampling from each subinterval equally helps ensure that the total sample is close to being uniformly-distributed over the entire population of K representable numbers. Although this method is not as random as if the exponents were also generated randomly, it is easier to use when testing numerical software with the n -bit simulator, since it is easier to construct the sample numbers. In this investigation, all pseudo-random samples were generated by combining a pseudo-random normalized mantissa (from the subinterval $[0.5, 1.0]$) with a pseudo-random exponent which was uniformly-distributed over the population of possible exponents. Resulting samples which fell outside the variable ranges were discarded.

Testing of Pseudo-Random Number Generators

Three considerations play important roles in determining whether or not a particular source provides uniformly-distributed random or pseudo-random numbers which are adequate for use in testing. First, the numbers must be able to pass statistical tests which reveal departures from uniformity and independence. Second, the numbers must be sufficiently dense over the interval being used, which in this case is the subinterval $[0.5, 1.0]$. Third, the numbers should be able to be produced efficiently. Since these three properties rarely characterize any one method of producing random numbers, compromises are made. Uniformity and independence are generally more important than density or efficiency in determining

the adequacy of any particular method (Ref 19:169-170). The numbers returned by the CDC pseudo-random number generator are uniformly-distributed over the interval $[0.0, 1.0]$ and can be shown to pass the pair triplet test, the auto-correlation test with lag ≤ 100 , and one of the most powerful tests, the spectral test formulated by R. R. Conveyou and R. D. MacPherson (Ref 38:82 and 14:82). The CDC pseudo-random number generator, when used in conjunction with the n-bit simulator and equation (22), can be used to efficiently produce numbers which are sufficiently dense over the interval $[0.5, 1.0]$. There are many tests designed to reveal departures from independence and a uniform distribution. Knuth (Ref 38) describes ten of them with algorithms, and other tests can be found in references 34, 45, and 54. The tests discussed here represent those that were used to test the CDC pseudo-random number generator when used in conjunction with the n-bit simulator. Tests were conducted for a uniform distribution, randomness, and correlation.

Uniform Distribution. The Kolmogorov-Smirnov test (Ref 19:187-188) was used to test whether or not pseudo-random numbers were uniformly-distributed over the interval $[0.5, 1.0]$. The sample cumulative distribution function is

$$F_n(X) = \begin{cases} 0 & X < X_1 \\ \frac{i}{n} & X_i \leq X < X_{i+1} \quad i=1, \dots, n-1 \\ 1 & X \geq X_n \end{cases} \quad (23)$$

and the theoretical cumulative distribution function is

$$F(X) = \begin{cases} 0 & X < 0.5 \\ 2X-1 & 0.5 \leq X < 1.0 \\ 1 & X \geq 1.0 \end{cases} \quad (24)$$

The test statistic $D_n(X)$ is

$$D_n(X) = \max_X |F(X) - F_n(X)| \quad (25)$$

The null hypothesis that the samples are taken from a uniform distribution may be rejected with a confidence of $1-\alpha$ if $D_n(X) \geq D_\alpha$, and if $D_n(X) < D_\alpha$, the null hypothesis cannot be rejected. For $\alpha = 0.10, 0.05$, and 0.01 , the critical D_α values were computed using equations (26), (27), and (28).

$$D_{.10} = \frac{1.22}{\sqrt{n}} \quad (26)$$

$$D_{.05} = \frac{1.36}{\sqrt{n}} \quad (27)$$

$$D_{.01} = \frac{1.63}{\sqrt{n}} \quad (28)$$

The results of Kolmogorov-Smirnov tests of pseudo-random numbers are shown in Table 1. Columns 1 and 2 show what the numbers were used for, column 3 shows the number of samples generated, and column 4 shows the computed D_n values. These can be compared to values shown in columns 5, 6, and 7 which show the critical D_α values used to test with a level of significance α . These numbers tested were used in evaluating the square root and sine function approximations which are discussed in the next chapter. When testing, odd-numbered calls to the pseudo-random number generator were used to generate the mantissa and the even-numbered calls were used to generate the exponents. The mantissas were modified, if needed, by using equation (22) and were then truncated to contain 23 significant bits, which is the single-precision mantissa length of the AN/AYK-15A flight computer. The numbers used to generate the exponents were tested as returned by the pseudo-random number generator. The reasons for choosing a particular number of samples to use will be explained later in this chapter and in the next chapter.

USE		NUMBER OF SAMPLES	$D_n(X)$	$D_{0.10}$
Square Root	Mantissa	600	0.0414	0.0498
	Exponent	600	0.0307	0.0498
Sine Cosine	Mantissa	25307	0.00470	0.00767
	Exponent	25307	0.00549	0.00767

Table 1. Results of Kolmogorov-Smirnov Tests

Independence. Randomness and correlation should be tested for when evaluating the independence of random numbers. Randomness for single numbers was tested by using the runs test.

Runs Test. The runs test (Ref 47:282-285) is based on the order in which samples are obtained. The median of the samples is obtained, and then each sample is compared to the median. Numbers less than the median are given the value 0 and numbers greater than the median are given the value 1. Numbers which equal the median are not considered. The number of runs of zeros and ones can be approximated closely by a normal distribution with mean μ_u and standard deviation σ_u as shown in equations (29) and (30).

$$\mu_u = \frac{2n_1n_2}{n_1+n_2} + 1 \quad (29)$$

$$\sigma_u = \sqrt{\frac{2n_1n_2(2n_1n_2 - n_1 - n_2)}{(n_1+n_2)^2(n_1+n_2-1)}} \quad (30)$$

The null hypothesis that the sample is random can be based on the statistic as shown in equation (31), where u is the observed number of runs

$$z = \frac{u - \mu_u}{\sigma_u} \quad (31)$$

It is assumed that no more than one value will exactly equal the median,

thus n_1 and n_2 can be computed by

$$n_1 = n_2 = \begin{cases} N/2 & N \text{ even} \\ (N-1)/2 & N \text{ odd} \end{cases} \quad (32)$$

With this assumption, μ_u and σ_u can be computed by

$$\mu_u = \begin{cases} (N+2)/2 & N \text{ even} \\ (N+1)/2 & N \text{ odd} \end{cases} \quad (33)$$

$$\sigma_u = \begin{cases} \sqrt{\frac{N(N-2)}{4(N-1)}} & N \text{ even} \\ \sqrt{\frac{(N-1)(N-3)}{4(N-2)}} & N \text{ odd} \end{cases} \quad (34)$$

The null hypothesis that the samples are random is rejected with a confidence of $1-\alpha$ if z falls outside the confidence interval $[-z_{\alpha/2}, z_{\alpha/2}]$. Results of tests using the same pseudo-random numbers used for the Kolmogorov-Smirnov test are shown in Table 2. Columns 1 and 2 show what the numbers were used for and column 3 shows the number of samples generated. Columns 4 and 5 show the values of the expected mean μ_u and the standard deviation σ_u . Column 6 shows the actual number of runs and column 7 shows the values of the z statistics computed using equation (31). These can be compared to the values of $z_{\alpha/2}$ (two-tailed test) for $\alpha = 0.1, 0.05$, and 0.01 .

Serial Correlation Test. The serial correlation test (Ref 38:64-65) measures the amount that U_{j+1} depends on U_j , where U is an individual sample. The serial correlation coefficient C is a statistic which always lies between -1 and $+1$ and is computed by

$$C = \frac{n(U_0 U_1 + U_1 U_2 + \dots + U_{n-2} U_{n-1} + U_{n-1} U_0) - (U_0 + U_1 + \dots + U_{n-1})^2}{n(U_0^2 + U_1^2 + \dots + U_{n-1}^2) - (U_0 + U_1 + \dots + U_{n-1})^2} \quad (35)$$

When C lies close to zero, then any two consecutive samples are rela-

USE		NUMBER OF SAMPLES	μ_u	σ_u	U	Z	$Z_{0.05}$
Square Root	Mantissa	600	301	12.237	290	0.899	1.645
	Exponent	600	301	12.237	288	1.062	1.645
Sine Cosine	Mantissa	25307	12654	79.538	12773	1.496	1.645
	Exponent	25307	12654	79.538	12701	0.591	1.645

Table 2. Results of Runs Test

USE		NUMBER OF SAMPLES	μ_n	σ_n	$\mu_n - 2\sigma_n$	C	$\mu_n + 2\sigma_n$
Square Root	Mantissa	600	-0.00167	-0.04076	0.07984	0.0234	-0.08318
	Exponent	600	-0.00167	-0.04076	0.07984	0.0287	-0.08318
Sine Cosine	Mantissa	25307	-0.00004	-0.00629	0.01253	-0.0148	-0.01261
	Exponent	25307	-0.00004	-0.00629	0.01253	-0.0008	-0.01261

Table 3. Results of Serial Correlation Test

tively independent of each other, but when C is ± 1 it indicates a total linear dependence, i.e. $U_j = m \pm aU_{j-1}$ for all j and for some constants a and m . "Good" values of C are expected to lie between $\mu_n - 2\sigma_n$ and $\mu_n + 2\sigma_n$ about 95 percent of the time, where μ_n and σ_n are computed by

$$\mu_n = \frac{-1}{(n-1)} \quad (36)$$

$$\sigma_n = \frac{-1}{(n-1)} \sqrt{\frac{n(n-3)}{n+1}} \quad (37)$$

These two equations are only conjectured by Knuth but are supported by empirical evidence (Ref 38:64-65). Successive mantissas and successive exponents were tested separately for correlation. Table 3 shows the results of tests using the same pseudo-random numbers used for the Kolmogorov-Smirnov test. Columns 1 and 2 show the use of numbers and column 3 shows the number of samples generated. Column 4 and 5 show the values of the expected mean μ_n and standard deviation σ_n . Column 6 shows the values of $\mu_n - 2\sigma_n$, column 7 shows the values of the C statistics computed using equation (35), and column 8 shows the values of $\mu_n + 2\sigma_n$.

Termination Criteria for Monte Carlo Testing

Exhaustive testing is prohibitive in time and cost in all but some special cases. The tester, if he decides to test using random input data, is therefore faced with a decision on when to stop testing. It is assumed that the software, in the state in which it is being tested, will be rejected if any error is encountered. When testing numerical properties, an error occurs whenever the relative or absolute error exceeds specified tolerance limits. In this sense, testing stops when the first error occurs. A test is considered to be the act of selecting a random sample and then determining if that sample produces an error or not. In this sense, all tests are assumed to be independent. For each test,

there are only two possible outcomes: an error occurs, or no error occurs. An error occurrence is considered to be a success and no error occurrence is considered to be a failure. It is also assumed that the probability of a success, p , is the same for all trials, or tests, since the test includes the selection of the random sample. If the first success occurs on the x th trial, it must be preceded by $x-1$ failures. The probability of $x-1$ failures is $(1-p)^{x-1}$, and if multiplied by the probability of a success on the x th trial, p , the probability of getting the first success on the x th trial is obtained as shown in equation (38) (Ref 47:54-55,83).

$$g(x;p) = p(1-p)^{x-1} \quad (38)$$

This is the geometric distribution, and for the purposes of this investigation, it is assumed that the testing of numerical software will follow this distribution.

If no errors occur, the tester is faced with the decision of when to stop testing. If it is assumed that the software will not be rejected unless an error is actually detected, then the producer assumes no risk, and therefore the probability of a Type I error is zero. The consumer's actual risk after n trials is the probability that the software was not rejected if it contained errors. For the software to have not been rejected, all n trials would have to produce failures. Since the probability of a failure on any one trial is $1-p$, the probability of not rejecting the software given n trials is $(1-p)^n$. The probability of a Type II error, β , is specified by the consumer (or tester) and is the maximum risk the tester is willing to take of accepting bad software. To ensure that the tester's actual risk is less than β , n must be chosen large enough so that $(1-p)^n \leq \beta$. Solving for n gives

$$n \geq \frac{\ln \beta}{\ln(1-p)} \quad (39)$$

To get a value for p , some assumptions must be made about numerical errors in software. The following example is presented to show why assumptions might be made to compute p . The AN/AYK-15A computer with its 23-bit mantissa is used in this example. If the function $\sin(\pi X/2)$ is being evaluated over the interval $[0.5, 1.0]$, then the exponent of all possible inputs is always 0, and the values of X are uniformly distributed across the interval. However, there are 2^{22} , or 4,194,304 of these values. If the software is rejected with only one error (and it is assumed that all the rest of the inputs do not produce errors), then p is the percent of the input tested defective to be used as the rejection criterion and may be computed by

$$p = \frac{1}{2^{22}} = \frac{1}{4194304} \approx 0.2384 \times 10^{-6} \quad (40)$$

If the tester wishes to be 95 percent sure he correctly identifies a piece of software for rejection, then β is 0.05. Using these values, the tester would have to conduct nearly three times as many tests with random samples as there are input possibilities. The value of n is computed by

$$n \geq \frac{\log \beta}{\log(1-p)} \approx \frac{-1.30103}{-0.10354 \times 10^{-6}} \approx 0.12565 \times 10^8 \quad (41)$$

This says it would be easier to conduct an exhaustive test. The problem lies in the assumption of the single error found among all possible inputs. It seems highly unlikely that one value would exceed the error tolerances without the two numbers adjacent to it also exhibiting similar characteristics. The problem becomes one of specifying a realistic value for p .

A simple case of cancellation of terms is presented to show how relative errors large enough to cause software rejection can occur. In this example, the binary machine truncates, the mantissa length is m , and the relative error bound is specified to be 2^{-k} . If, for example, the relative error bound is specified to be 10^{-5} , then $k = 5(\ln 10)/(\ln 2)$, or approximately 16.61. The variable Y is computed as shown in equation (42), and the constant C , which has an exponent e , is assumed to have a relative error of 2^{-m} .

$$Y = C - X \quad (42)$$

For all values of X within a distance D of C where D is computed by

$$D = (2^{-m+k-1}) * (2^e) \quad (43)$$

at least the $(m-k+1)$ most significant bits of the mantissa are lost in the subtraction, leaving at most $k-1$ significant bits. If it is assumed that all values of X which fall within a distance D of C have the same exponent as C , then the number of unique values of X which lie within a distance D of C is computed to be Z by

$$Z = 1 + 2*(2^{k-1}-1) = 2^k - 1 \quad (44)$$

Since the total number of possibilities for X (with the same exponent as C) is 2^{m-1} , the portion of the subinterval which causes rejection is R .

$$R = Z/(2^{m-1}) = \frac{2^k - 1}{2^{m-1}} \quad (45)$$

For the case of the AN/AYK-15A computer, if k has been specified to be 17 (giving 10^{-5} relative accuracy), then R may be computed by

$$R = \frac{2^{17} - 1}{2^{23-1}} \approx 2^{-5} = 0.03125 \quad (46)$$

This value of R says that there is a block of consecutive numbers centered about C in one subinterval which cause a relative error greater

than 2^{-17} , and that these numbers comprise over three percent of the numbers in that subinterval.

In actual situations, it cannot be assumed that all the numbers within a distance D of C have the same exponent as C , so Z' must be computed. If the exponent of C is neither the maximum positive nor negative exponent of the computer being simulated, then for binary computers, Z' falls in the range $[3Z/4, 3Z/2]$. If the exponent is the maximum positive value, then Z' falls in the range $[Z/2, 3Z/2]$, and if the exponent is the maximum negative value, then Z' falls in the range $[Z/2, Z]$. In these special cases, overflow and underflow combined with the normalization of the mantissa can limit the number of unique values within a distance D of C , so Z' can take on smaller values. If R^* is computed as shown in equation (47), then R^* gives a minimum proportion of consecutive numbers in the same subinterval as the value C which cause too large a relative error.

$$R^* = \min(Z')/2^{m-1} = \frac{Z}{2}/2^{m-1} = \frac{Z}{2^m} = (2^k - 1) * (2^{-m}) \quad (47)$$

If the gross assumption is made that relative errors 1) propagate through a piece of software, 2) generally don't diminish, and 3) are caused by cancellation of terms, then R^* can be used to approximate p in equation (39).

$$n \geq \frac{\ln \beta}{\ln(1-p)} \quad (39)$$

If the tester is testing over w subintervals, then the probability of getting a success is p/w and equation (39) can be rewritten to reflect the testing over the entire variable range.

$$n \geq \frac{\ln \beta}{\ln(1-\frac{p}{w})} \quad (48)$$

When a multi-variate function is being evaluated, another assumption is made concerning the proportions of different variables which can cause a success (an error larger than the specified limits) to occur. It is assumed that a certain proportion of each variable cause success and that these proportions are independent of the other variables. Certain combinations of input values, none of which alone causes a success, can together also cause a success to occur. However, these cases were not considered in arriving at the number of random x trials needed, since they would tend to decrease the number of trials required. As in the single-variate case, each variable will be sampled from its total number of possibilities using pseudo-random numbers to generate the mantissa and exponent. The proportion of all possible values of the i^{th} variable which cause success is represented by p_i with the range of the i^{th} variable being broken up into w_i subintervals. The total number of variables is denoted by V . The proportion P of all possible multi-variate inputs is computed by

$$P = \sum_{i=1}^V \frac{p_i}{w_i} - \sum_{\substack{i,j \\ i \neq j}}^V \frac{p_i}{w_i} \frac{p_j}{w_j} + 0^3 \quad (49)$$

where 0^3 denotes terms of order three or higher.

The number of input samples (n) which guarantee that the consumer's risk is not greater than β can then be obtained using

$$n \geq \frac{\ln \beta}{\ln(1-P)} \quad (50)$$

This still does not account for gradual error buildup in which a much smaller proportion of a subinterval produces a relative error which may be only one bit short of the required relative accuracy. In these cases, the error buildup can usually be noticed using error plots, since many

other numbers might provide the required relative accuracy with as few as zero or one bits to spare. Cody (Ref 10) suggest that when this occurs and it appears likely that the software will fail, more extensive testing can be conducted over the appropriate subintervals in question.

Anomalies other than cancellation of terms can occur in numerical software. Cody (Ref 10) also suggests that special inputs be constructed which will check the boundaries of subintervals, boundaries of the input variable, and any cross-over and neighboring data points where the routine changes algorithms. These cases are difficult to construct and must currently be done manually. When used in conjunction with random testing using subintervals, they increase the tester's confidence that a numerical software routine performs correctly.

Summary

A tester using Monte Carlo techniques to test software is faced with the question of when to stop testing. His maximum risk, β , is specified, but he may know very little about the error characteristics of the software. By making an assumption about the nature of the error characteristics, the tester can arrive at a number of trials at which to stop testing. The tester's confidence in the results (assuming the tester doesn't get any successes using random inputs) can be increased by conducting extra tests at the boundaries of the variables and other special points, by using error plots, and by monitoring the error deviations over small subintervals.

IV Evaluation of Avionics Mathematical Routines

Many inertial navigation and fire control routines require that cosine, sine, arctangent, and square root functions be evaluated. Two basic methods were considered for simulating the trigonometric functions and the square root function when they are required by a simulated flight routine. The Control Data Corporation (CDC) library routines could be used, with the simulator adjusting the function value returned, or new routines could be constructed which would then be modified by the n-bit simulator. These modified routines would then be compiled along with the programs which call them, thereby more accurately reflecting the accuracy obtainable on the simulated machine. The sine, cosine, arctangent, and square root functions were all evaluated with the transmitted error being removed (Ref 11:129). The arctangent function, when required by a simulated flight routine, was evaluated using the CDC library routine with the result being truncated by the n-bit simulator. Other arctangent approximations were not analyzed. Several have been proposed by Hart (Ref 32). The algorithms presented in this chapter for the square root, sine, and cosine functions are "optimal" for the AN/AYK-15A flight computer in the sense that they provide the required accuracy specified by reference 57 while requiring a minimum number of computer instructions for implementation. Other approximations which were analyzed are presented in Appendix C. All stored constants have been chosen to minimize the relative error, since this is the same as maximizing the number of correct mantissa bits (mantissa normalized). In evaluating these functions, the absolute error, AE, is defined as the difference between the approximated solution, W, and the exact solution, Y.

$$AE = W - Y \quad (51)$$

The relative error, RE, is defined to be the absolute error divided by the exact solution and is not defined when the exact solution equals zero (Ref 18:5);

$$RE = AE/Y = (W - Y)/Y \quad (Y \neq 0) \quad (52)$$

For the plots shown in this chapter, the horizontal axes are used to represent the input arguments and the vertical axes are used to represent either the absolute error, AE, or the relative error, RE.

Square Root

For the square root function, several approximations were analyzed in addition to that of truncating the CDC result with the n-bit simulator. The two solutions proposed both use Newton's iterative method for evaluating \sqrt{X} where $X > 0$ (Ref 18:23). Newton's method is to choose an initial approximation Y_0 and then compute Y_1, Y_2, Y_3, \dots defined by the recursion formula

$$Y_{k+1} = \frac{1}{2} \left(Y_k + \frac{X}{Y_k} \right), \quad k = 0, 1, 2, \dots \quad (53)$$

Convergence is quadratic, meaning that as Y_k gets sufficiently close to \sqrt{X} , Y_{k+1} has approximately twice as many correct digits as Y_k . Range reduction schemes are commonly employed to reduce the maximum relative error in the first approximation.

Specifications. Specifications for the square root function include the following:

- 1) that it not call other subroutines (Ref 23:58 and 57:149)
- 2) that the maximum output would be 3276.7 (Ref 57:92)
- 3) that the maximum error was to be 0.01

The number 3276.7 is the maximum aircraft speed which might be encountered when computed by

$$V_a = \sqrt{V_x^2 + V_y^2} \quad (54)$$

V_x and V_y represent aircraft velocity vectors in an X, Y coordinate system, and V_a represents the aircraft track vector with maximum magnitude 3276.7. The maximum error of 0.01 corresponds to the resolution of the aircraft instruments and was assumed to be the maximum absolute error which would occur at or near the maximum output value. The maximum relative error, MRE, was therefore assumed to be 3.0×10^{-6} ;

$$\text{MRE} = 3.0 \times 10^{-6} < 3.052 \times 10^{-6} \approx \frac{0.01}{3276.7} \quad (55)$$

Assumptions. For the purpose of measuring absolute and relative errors which were estimated to be on the order of 10^{-6} , the CDC square root function with its relative error bound of 3.0×10^{-16} (Ref 14:139) was assumed to be accurate enough to use as the standard.

Testing Criteria. Each solution was evaluated using Monte Carlo techniques as described in the previous chapter. β was chosen to be 0.01 and only two subintervals were tested: $[0.25, 0.50]$ and $[0.50, 1.00]$. This was because only the relative error was being measured. The relative accuracy required was 10^{-5} , or approximately 2^{-17} . Using values of 17 for k and 23 for m (mantissa length), R^* is computed to be 0.015625,

$$R^* = (2^k - 1) * (2^{-m}) \quad (56)$$

This value is then substituted for p in

$$n \geq \frac{\ln \beta}{\ln(1 - \frac{p}{w})} \quad (57)$$

to obtain a value for n, where n is the number of pseudo-random samples required to test to a significance level of $1 - \beta$ (see Chapter 3).

The value of n is computed to be 588;

$$n \geq \frac{\ln(0.01)}{\ln(1 - \frac{0.015625}{2})} \approx 587.2 \quad (58)$$

The results of testing 600 random trial numbers for uniformity, randomness, and correlation are shown in Tables 1, 2, and 3 of the preceding chapter. These numbers were supplemented by 10,000 other pseudo-random numbers when constructing the absolute and relative error plots. The extra numbers were used to smooth out the error curves. The purpose in smoothing out the error curves was to show more accurately the nature of the maximum error. These extra numbers were generated by single calls to the CDC pseudo-random number generator. Each call returns a value uniformly-distributed over the interval $[0.0, 1.0]$, and by modifying those numbers which fall in the interval $[0.00, 0.25]$ by

$$S \leftarrow 0.5 - S \quad (59)$$

approximately 5000 pseudo-random numbers were obtained for each of the two subintervals being tested. All the inputs were divided into 100 evenly-spaced intervals for plotting purposes. The plots show the maximum (positive and negative) absolute and relative errors obtained over each of the 100 plotting intervals.

Solution 1. The first method for approximating the square root function is to call the CDC square root function and truncate the result to the desired wordlength. If the computer being simulated has a mantissa length of m bits, then the effect of truncating a value such as that returned from the CDC square root routine is to introduce a negative relative error (see equation 52) not less than -2^{-m+1} . For the AN/AYK-15A flight computer, this relative error bound would be -2^{-22} ; or approximately -2.4×10^{-7} . The absolute and relative error plots over the interval $[0.00, 1.00]$ are shown in Fig. 9 and Fig. 10, with each plot

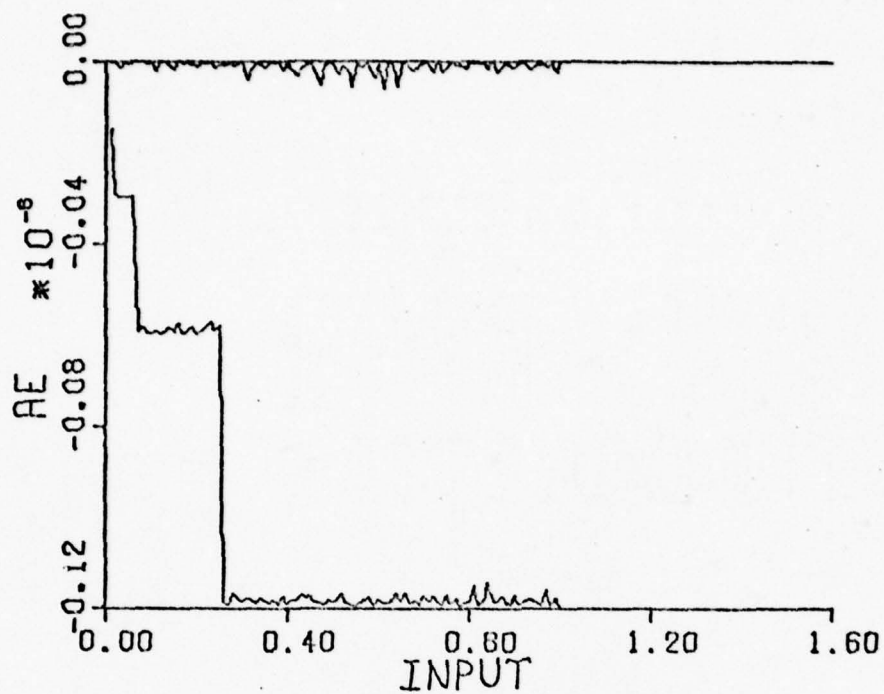


Fig. 9. Absolute Error of Truncated CDC Square Root

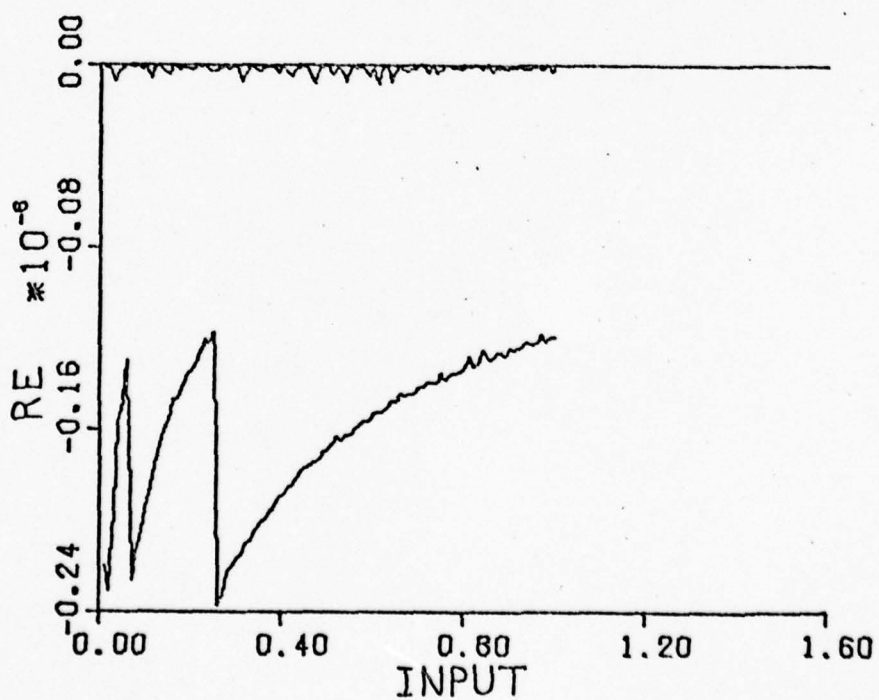


Fig. 10. Relative Error of Truncated CDC Square Root

showing the maxima and minima over the 100 evenly-spaced intervals. Each break in the lower bound occurs where the exponent of the square root changes.

Using the CDC square root function is easier than using either of the other two solutions which follow, since only one line of code is required in the FORTRAN flight routine. The error bounds are approximately the same as those produced by solutions 2 and 3, so little can be gained (in terms of simulation accuracy) by using either solution 2 or 3 when simulating.

Solution 2. The second method for computing the square root function uses a combination of a linear minimax polynomial and an exponent shift to get an initial approximation (Ref 33:25). Once the first approximation has been computed, two iterations of Newton's method are applied to obtain the desired relative accuracy.

The exponent of the initial approximation is computed by right-shifting one place the exponent of the input argument. If a 1 is shifted off (odd exponent), then the shifted exponent is increased by one and the mantissa is shifted one place to the right. A relative error of -2^{-23} can be introduced if, during a mantissa shift, a 1 is shifted off the end. This mantissa shift then leaves an unnormalized mantissa which lies in the range $[0.25, 0.50]$. Otherwise the mantissa lies in the normal range $[0.50, 1.00]$. A linear minimax polynomial,

$$M_0 = A*M + B \quad (60)$$

is used as the first approximation, M_0 , to the mantissa of the square root of the input argument with mantissa M . This is then combined with the shifted exponent to obtain the first approximation Y_0 for the square root of X .

Since the error measured is the relative error, the minimax property (that the maximum error is a minimum) should hold for the relative error of the final output value. Hemker, et al (Ref 33:25), solve for the two minimax coefficients A and B, giving $A = 0.6862915010151$ and $B = 0.3431457505076$. These coefficients may also be expressed by

$$A = 2^*R \quad (61)$$

$$B = 6-4 \sqrt{2} \quad (62)$$

The maximum relative error, MRE of the initial approximation, occurs at both endpoints and at one interior point and may be computed using the right endpoint;

$$MRE = \max \left| \frac{A*X+B-\sqrt{X}}{\sqrt{X}} \right| = \left| \frac{A+B-1}{1} \right|_{X=1} = 17-12 \sqrt{2} \approx 0.02944 \quad (63)$$

Two iterations of Newton's method carried out on a machine with infinite precision would give a final maximum relative error less than $9*10^{-8}$. This is only slightly better than the actual relative accuracy obtainable using single precision (23-bit mantissa) on the computer being simulated.

Solution 2 was tested over the interval $[0.25, 1.00]$ with the numbers in the interval $[0.25, 0.50]$ representing those that would have had an odd exponent. Hence, their accuracy was truncated to 22 bits, thereby simulating the extra right shift. The minimax polynomial and the two Newton iterations were evaluated using a 23-bit mantissa for all cases. The absolute and relative error plots are shown in Fig. 11 and Fig. 12.

The effect of the initial minimax approximation for the mantissa can still be observed in these plots. If another iteration of Newton's method is performed on the AN/AYK-15A computer, this effect would completely disappear. However, the maximum (negative) relative error bound

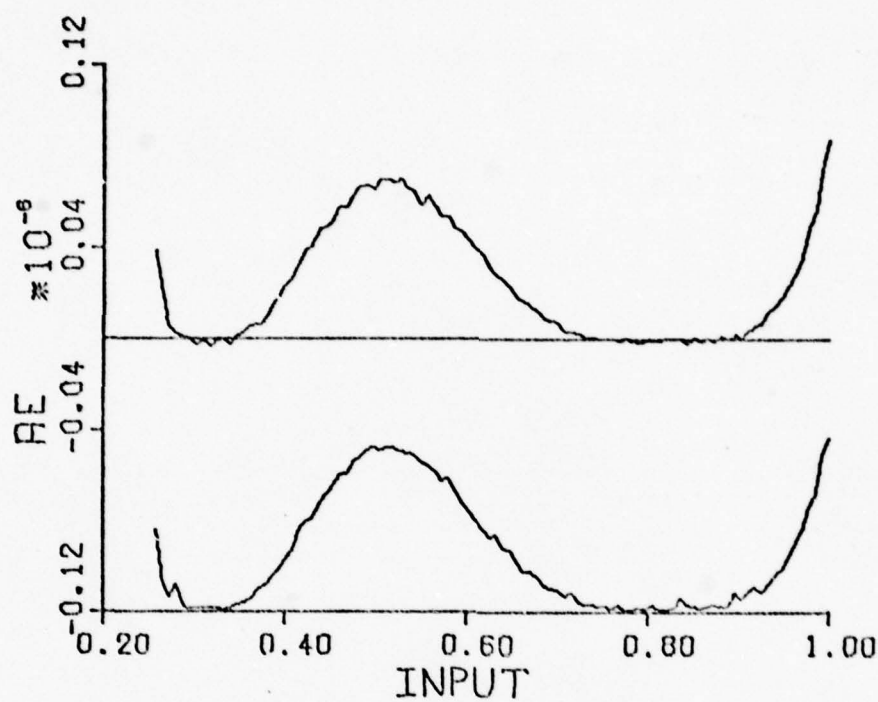


Fig. 11. Absolute Error of Square Root Solution 2

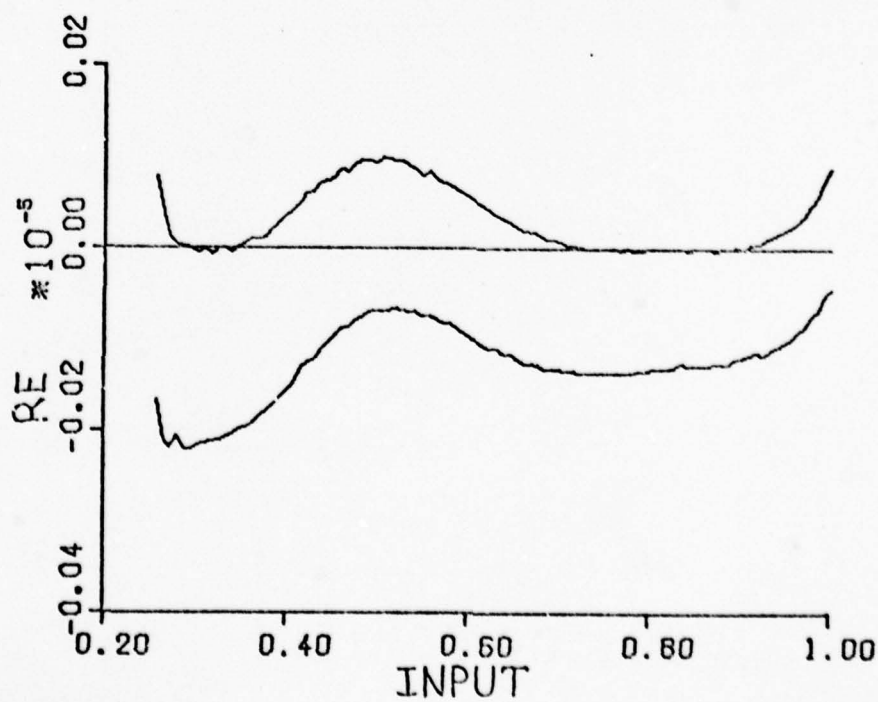


Fig. 12. Relative Error of Square Root Solution 2

would not be improved any.

Solution 3. The third method, which also uses a combination of a linear minimax polynomial and an exponent shift to get an initial approximation, is presented as an alternative to the second solution. The exponent of the first approximation is computed by right-shifting one place the exponent of the input argument. If a 1 is shifted off (odd exponent), this method sets a flag to signal a later multiplication by a pre-stored constant $\sqrt{2}$. No mantissa shifting is required, so the mantissa always lies in the range $[0.50, 1.00]$. Since the range for the minimax approximation is smaller, the maximum error using the linear polynomial is much smaller. The number resulting from the exponent shift combined with the mantissa from the minimax polynomial is then multiplied by $\sqrt{2}$ if the exponent of the input argument was odd (flag set). The coefficients of the minimax polynomial,

$$M_0 = A * M + B \quad (60)$$

may be expressed by

$$A = \sqrt{2} B \quad (64)$$

$$B = \frac{2}{(1 + \sqrt[4]{2})^2} \quad (65)$$

A and B, rounded to ten significant digits, have the values $A = 0.5901620671$ and $B = 0.4173075996$. The maximum relative error, MRE, still occurs at $X = 1.0$ and two other places and may be expressed by

$$\text{MRE} = \max \left| \frac{A * X + B - \sqrt{X}}{\sqrt{X}} \right| = \left| \frac{A + B - 1}{1} \right|_{x=1} = \left(\frac{1 - \sqrt[4]{2}}{1 + \sqrt[4]{2}} \right)^2 < 0.00747 \quad (66)$$

The maximum relative error, MRE, as shown does not account for the error introduced by the extra multiplication by $\sqrt{2}$ which is sometimes needed.

The pre-stored constant $\sqrt{2}$ has a relative error less than 2^{-23} , so to

get the maximum relative error, MRE' , of the initial approximation, the two relative errors must therefore be added.

$$MRE' = MRE + 2^{-23} < 0.00747 \quad (67)$$

This solution was tested over the interval $[0.25, 1.00]$ with the numbers in the interval $[0.25, 0.50]$ representing those inputs that would have had an odd exponent (and hence would have been multiplied by $\sqrt{2}$). The absolute and relative error plots are shown in Fig. 13 and Fig. 14.

This method has two distinct advantages over the method of Solution 2. First, it does not work with unnormalized numbers, and second, the initial estimate is much better than that for solution 2, thereby giving better convergence using Newton's method. The major disadvantage to this solution is the extra multiplication required whenever the exponent of the input argument is odd. Both solution 2 and solution 3 should be coded in an assembly language to facilitate the shifting required to get the initial approximation.

Recommendations. Two recommendations should be made concerning these three tests. First, when simulating using the n-bit simulator, it is easiest to just call the CDC square root function and truncate the answer to the appropriate accuracy. This also gives an adequate portrayal of the errors which would be encountered normally.

Second, when comparing the methods presented in solutions 2 and 3, the machine wordlength must be considered. The accuracy of the method presented in solution 2 is limited by the machine wordlength until a mantissa length of 24 or 25 bits is reached. For longer mantissa lengths, no significant accuracy is gained since the maximum relative error bound using only two iterations of Newton's method is approximately $9 \cdot 10^{-8}$, or $2^{-(23.4)}$. For mantissa lengths longer than 24 bits, solution

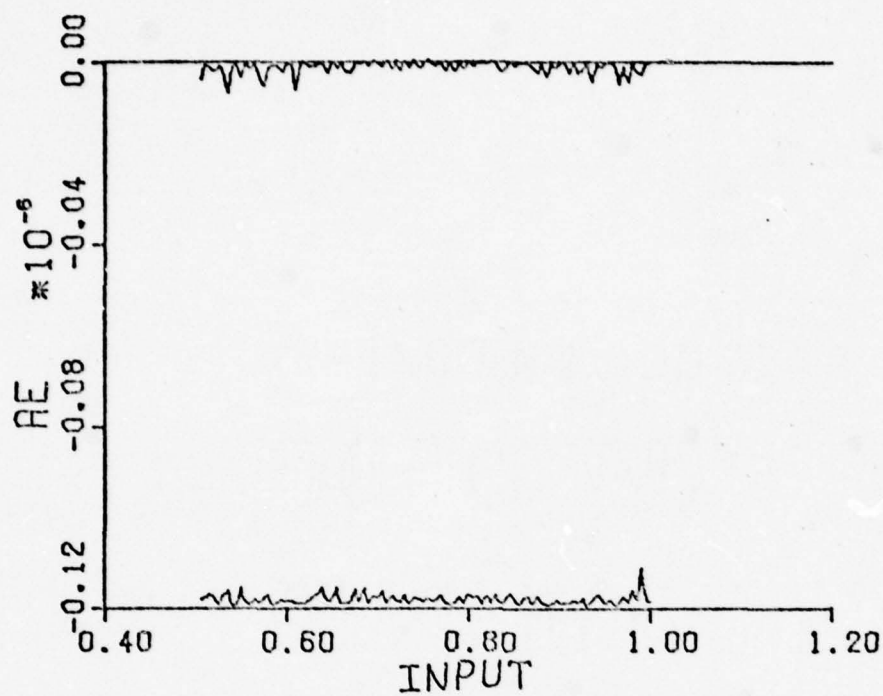


Fig. 13. Absolute Error of Square Root Solution 3

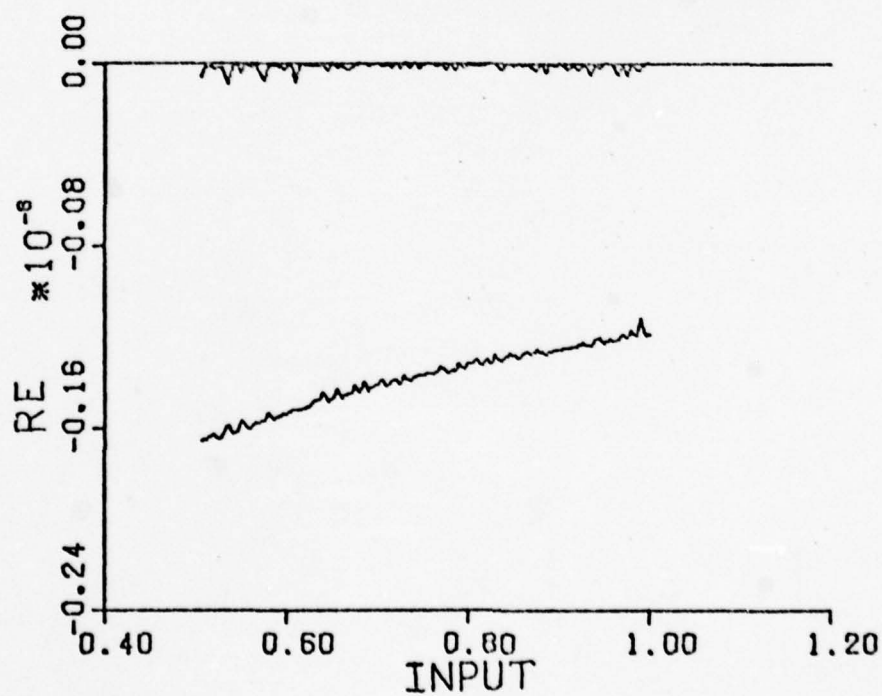


Fig. 14. Relative Error of Square Root Solution 3

2 could be modified to add more iterations of Newton's method. The accuracy of the method presented in solution 3 surpasses that of the method presented in solution 2 around the 24-bit mantissa length. The accuracy of solution 3 continues to improve significantly until a mantissa length of approximately 32 bits is reached, at which time the dominant error is no longer the machine truncation error. The reason that the method of solution 3 is able to obtain better accuracy lies in the fact that a shorter range is considered when using the minimax first approximation, thereby giving a much better result from the two Newton's method iterations.

Sine Function

For the sine function, several Taylor series, minimax, and continued fraction approximations were tested. Each proposed solution was executed using the n-bit simulator with single precision (23-bit mantissa) being specified.

Since errors on the order of 10^{-5} to 10^{-7} were expected, the CDC sine function, which has an error less than 10^{-15} (Ref 14:141), was considered to be exact in computing the absolute and relative errors of the proposed solutions. The formula for computing the absolute error AE is

$$AE = W - Y \quad (68)$$

and the formula for computing the relative error RE is

$$RE = AE/Y, (Y \neq 0) \quad (69)$$

In these equations, the variable W represents the answer obtained using the proposed solution and the variable Y represents the answer obtained from the CDC library sine function.

Specifications. Several conditions for the sine function were specified by the F-16 Inertial Navigations Specification (Ref 57) and

the F-16 Fire Control Specification (Ref 23). The first condition was that the data input range was to be $[-2\pi, 2\pi]$. The second condition was that the error was to be less than 10^{-5} . There was no indication whether this error bound was for the absolute error, the relative error, or both, so it was assumed to be the absolute bound for both. The third condition was that the sine function must not call other subroutines.

Assumptions. Several conditions which were assumed to be advantageous are shown below and are some of those listed by Hemker, et. al. (Ref 33:39-42):

- 1) $\lim_{X \rightarrow 0} \frac{\sin(X)}{X} = 1$
- 2) $\left| \frac{\sin(X)}{X} \right| \leq 1 \quad (\text{all } X)$
- 3) Optimal relative accuracy is obtained
- 4) The number of multiplications used is minimal
- 5) The odd character of the sine function is preserved.

Range Reductions. The primary reason for reducing the argument range of an approximation is to enable approximations to be used which have a small number of terms and can be evaluated quickly. For polynomial approximations, larger variable ranges mean higher order polynomials must be used to obtain the desired accuracy. Beyond some point, however, the costliness of the range reductions will offset the advantages (Ref 18:39,44).

The first range reduction for the sine approximation used the trigonometric identity

$$\sin(X) = -\sin(-X) \quad (70)$$

to reduce the range to $[0, 2\pi]$. This range reduction also guaranteed that the odd character of the sine function would be preserved. The

identity

$$\sin(X) = -\sin(2\pi - X) \quad (71)$$

was used to reduce the range further to $[0, \pi]$, and

$$\sin(X) = \sin(\pi - X) \quad (72)$$

was used to complete the range reduction, resulting in a final input range of $[0, \pi/2]$. All range reductions were accomplished using extended precision to reduce the effects caused by cancellation of terms. Since the sine approximation must not call other subroutines, the input range was not further reduced to $[0, \pi/4]$ using

$$\sin(X) = \cos(\pi/2 - X) \quad (73)$$

Testing Criteria. Solutions were evaluated using Monte Carlo techniques as described in the previous chapter. β was chosen to be 0.05 and for the AN/AYK-15A computer, there are 264 subintervals containing numbers from the interval $[-2\pi, 2\pi]$. Using values of 17 for k (2^{-k} accuracy - see Ref 23,57) and 23 for m (mantissa length), R^* is computed to be 0.015625;

$$R^* = (2^k - 1)(2^{-m}) \approx 0.015625 \quad (74)$$

This value is then substituted for p in

$$n \geq \frac{\ln \beta}{\ln(1 - \frac{p}{w})} \quad (75)$$

to obtain a value for n , where n is the number of pseudo-random samples required to test to a significance level of $1-\beta$ (see chapter 3).

The value of n is computed to be 50,615;

$$n \geq \frac{\ln(0.05)}{\ln(1 - \frac{0.015625}{264})} \approx 50614.4 \quad (76)$$

Since the interval $[0, 2\pi]$ only contains numbers from half as many intervals as the interval $[-2\pi, 2\pi]$ does, actual testing occurred over the

interval $[0, 2\pi]$. The first range reduction in the sine routine uses the trigonometric identity

$$\sin(X) = -\sin(-X) \quad (70)$$

so the test is just as significant over the reduced range. This allows w to be set to 132 in equation (69), and now n is computed to be 25307,

$$n \geq \frac{\ln(0.05)}{\ln(1 - \frac{0.015625}{132})} \approx 25306.4 \quad (77)$$

thus giving a 50 percent savings.

25307 pseudo-random trials were constructed and the results of testing these numbers for a uniform distribution, randomness, and correlation are shown in Tables 1, 2, and 3 of the preceding chapter. Usually, when such large samples are tested, the serial correlation test can be rewritten to be more efficient (Ref 38:64-65) and the Kolmogorov-Smirnov and Runs tests can be handled by establishing a chained list in increasing order of magnitude instead of using arrays held and sorted in core (Ref 19:188).

Solution. The solution presented is a seventh-order minimax approximation to the sine function with the relative error exhibiting the minimax property. Once the argument X has been reduced to the range $[0, \pi/2]$, it is scaled (not reduced) to the range $[0, 1]$ by the change of variables

$$Y = (2/\pi) * X \quad (78)$$

so that the sine function is now expressed in terms of the variable Y . The next step is to compute $Z = Y * Y$, thereby reducing the number of multiplications required. The minimax polynomial can then be expressed using Horner's Rule for nested multiplication;

$$\sin(\pi Y/2) = (((C_4 * Z + C_3) * Z + C_2) * Z + C_1) * Y \quad (79)$$

The coefficients of this polynomial were computed by Hart, et. al.

(Ref 32:117-118, 237) and are shown in the second column of Table 4. These coefficients were rounded to 23-bit accuracy and input in octal format. The approximate decimal values of the rounded coefficients are shown in the third column of Table 4.

	Coefficients Computed by Hart	Coefficients Rounded to 23-bit Accuracy
C_1	1.570794851	1.570794820786
C_2	-0.6459209764	-0.6459209918976
C_3	0.0794876547	0.07948765158653
C_4	-0.004362469	-0.004362468607724

Table 4. Minimax Sine Coefficients

Absolute and relative error plots for the approximation to the sine function on the reduced interval $[0, \pi/2]$ are shown in Fig. 15 and Fig. 16. The pattern which appears over this interval is found to repeat in various forms over the other parts of the interval $[0, 2\pi]$ for both sine and cosine approximations. This mirror image effect is caused by the range reductions. The absolute and relative error plots for the minimax approximation to the sine function on the interval $[0, 2\pi]$ (using the mentioned range reduction techniques) are shown in Fig. 17 and Fig. 18. These plots can be compared to the absolute and relative error plots shown in Fig. 19 and Fig. 20 which were obtained by truncating the value returned from the CDC 6600 library subroutine and using that as the approximation. As can be seen, the differences in the size of the absolute and relative errors are on the order of one magnitude. Because of the difference in the size of the errors, the seventh-order minimax polynomial was used when analyzing the flight routine in chapter 5, thereby ensuring that generated and analytic errors would be accurately

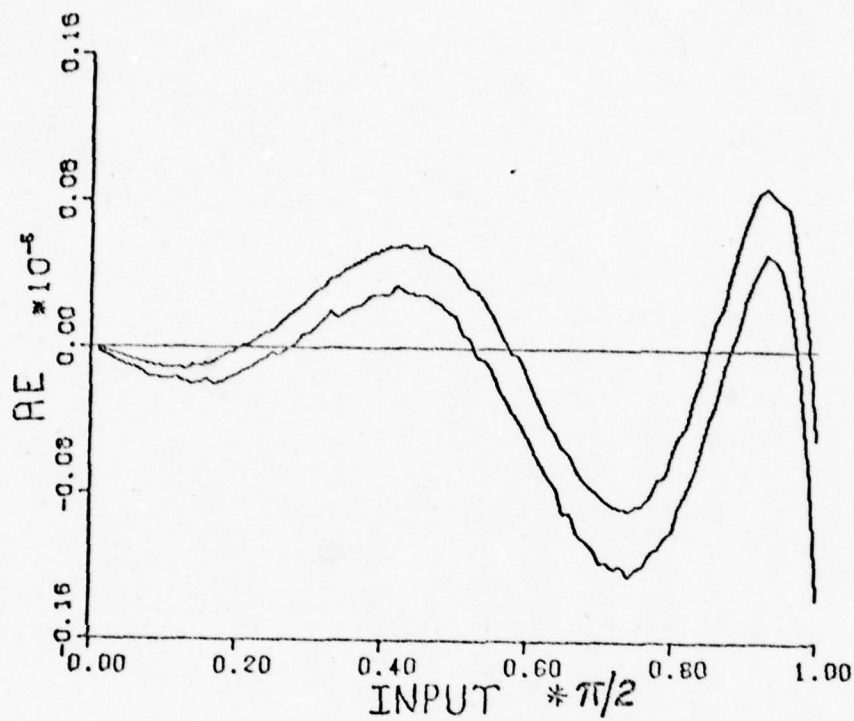


Fig. 15. Absolute Error of Sine Approximation on $[0, \pi/2]$

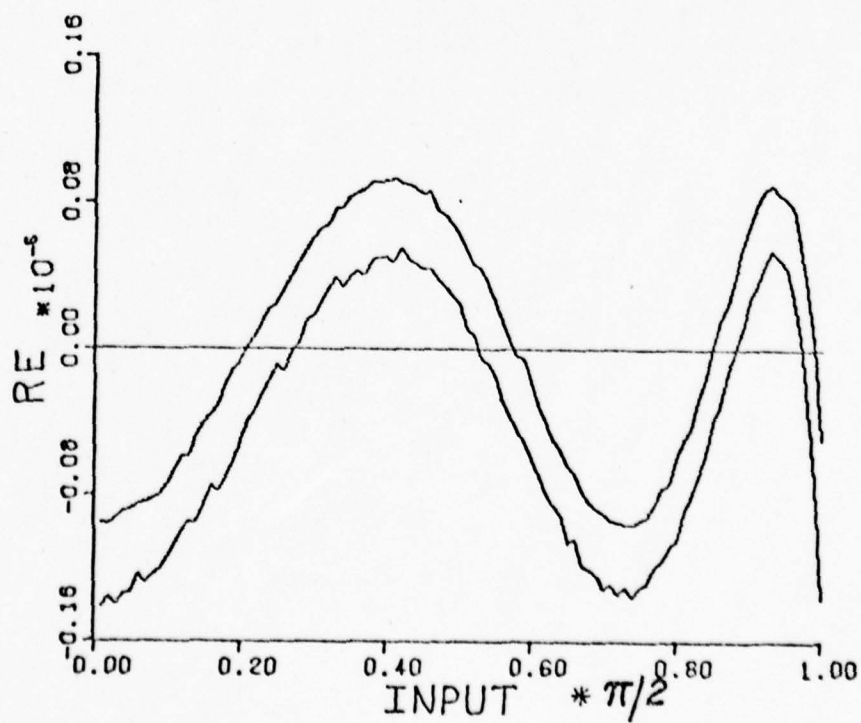


Fig. 16. Relative Error of Sine Approximation on $[0, \pi/2]$

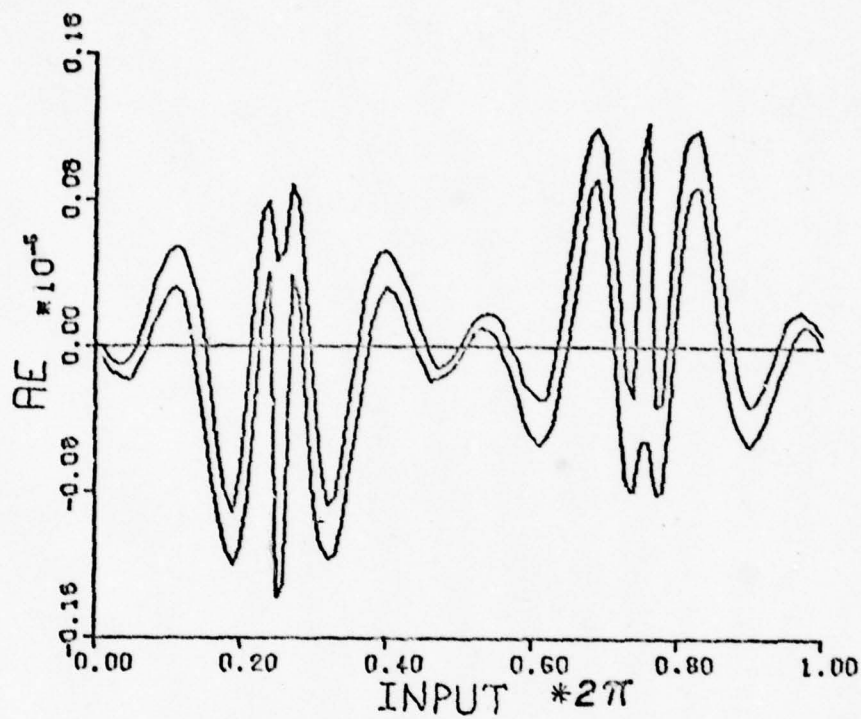


Fig. 17. Absolute Error of Sine Approximation on $[0, 2\pi]$

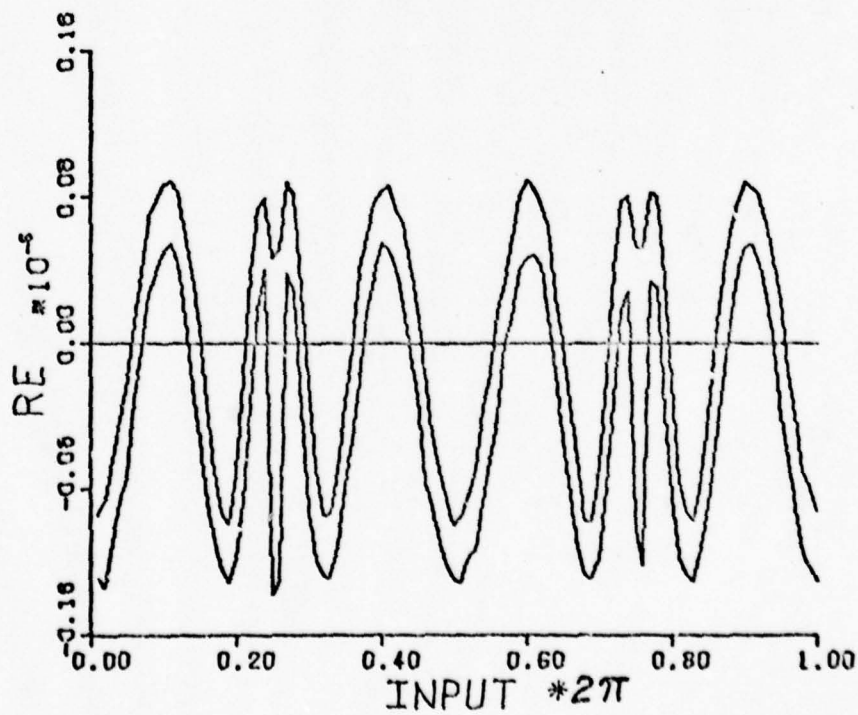


Fig. 18. Relative Error of Sine Approximation on $[0, 2\pi]$

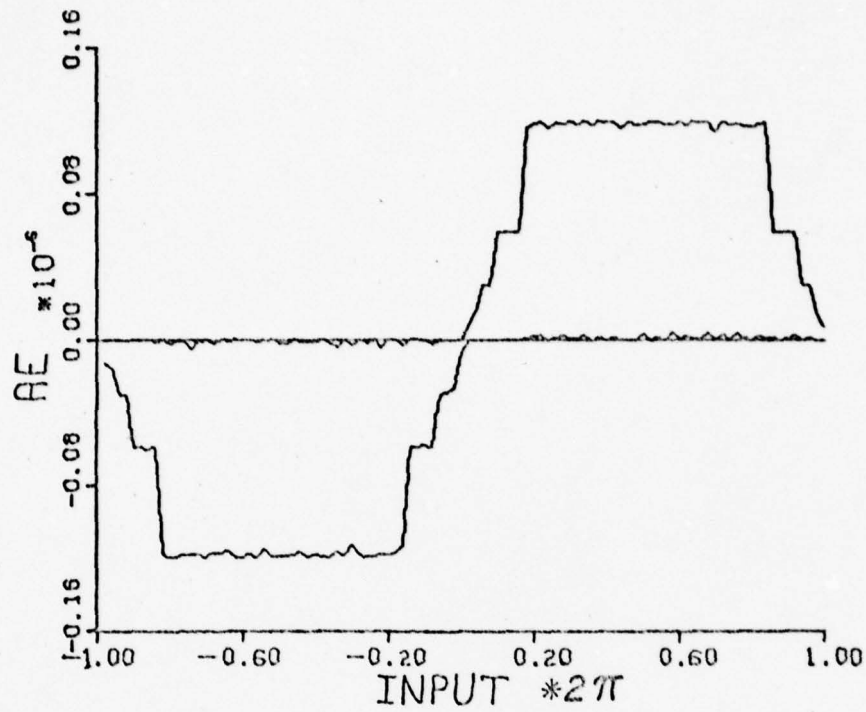


Fig. 19. Absolute Error of Truncated CDC Sine on $[-2\pi, 2\pi]$

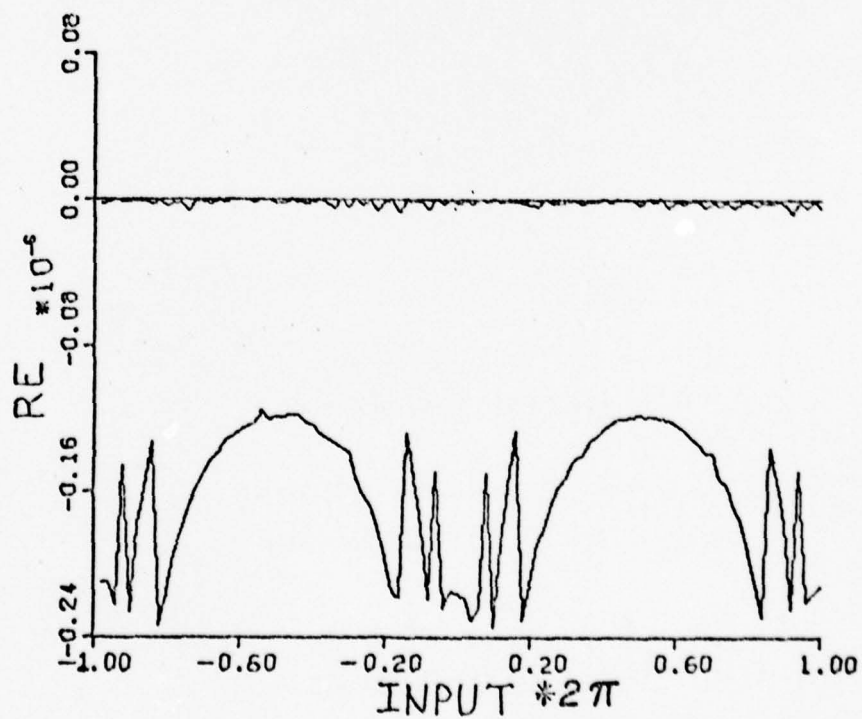


Fig. 20. Relative Error of Truncated CDC Sine on $[-2\pi, 2\pi]$

represented.

Summary. When constructing polynomial approximations to the sine function, range reduction techniques using trigonometric identities are employed. Once the argument range has been reduced, minimax polynomials can be used to minimize the maximum error (either absolute or relative). Since the sine function can be used in both additions and multiplications, both the absolute error and the relative error must be considered. Error plots can be used to aid visually in determining the error characteristics of the approximation.

Cosine Function

Since the sine and cosine functions are so closely related, ($\cos(X) = \sin(\pi/2 - X)$), only one routine with two entry points is needed. Whenever $\cos(X)$ is requested, $\sin(\pi/2 - X)$ can be evaluated. One minimax polynomial is used in the routine, and the different entry points facilitated coding for argument reduction.

Specifications. The specifications for the cosine function are the same as those for the sine function. As for the sine function, the error bound of 10^{-5} was assumed to be the bound for both absolute and relative errors.

Assumptions. Several conditions which were assumed to be advantageous are shown below and are some of those listed by Hemker, et. al. (Ref 33:39-42):

$$1) \lim_{X \rightarrow \frac{\pi}{2}} \left(\frac{\cos(X)}{\frac{\pi}{2} - X} \right) = 1$$

- 2) Optimal relative accuracy is obtained
- 3) The number of multiplications used is minimal
- 4) The even character of the cosine function be preserved.

Range Reductions. The first range reduction for the cosine approximation used the trigonometric identity

$$\cos(X) = \cos(-X) \quad (80)$$

to reduce the range to $[0, 2\pi]$. This also guaranteed that the even character of the cosine function would be preserved. The identity

$$\cos(X) = \cos(2\pi - X) \quad (81)$$

was used to reduce further the range to $[0, \pi]$, and

$$\cos(X) = -\cos(\pi - X) \quad (82)$$

was used to complete the range reduction, resulting in a final input range of $[0, \pi/2]$. Hemker's first condition for the cosine function meant that a power series approximation which is expanded about zero could not be used near $\pi/2$, since the relative error exceeds 10^{-5} . A further range reduction

$$\cos(X) = \sin(\pi/2 - X) \quad (83)$$

allows the minimax polynomial for the sine function to compute the cosine function also. This has the benefit of requiring only one routine with two entry points, one minimax polynomial, and one set of coefficients to compute both the sine and cosine functions.

Testing Criteria. The cosine approximation was tested using the same criteria as for the sine approximations. The same numbers were constructed over the interval $[0, 2\pi]$, and value of β was 0.05.

Solution. The cosine approximation is computed with the same minimax polynomial used to compute the sine approximation. The absolute and relative error plots for the minimax approximation to the cosine function on the interval $[0, 2\pi]$ (using the mentioned range reduction techniques) are shown in Fig. 21 and Fig. 22. Just as for the sine approximation, the pattern of the minimax polynomial can be seen to repeat, with the

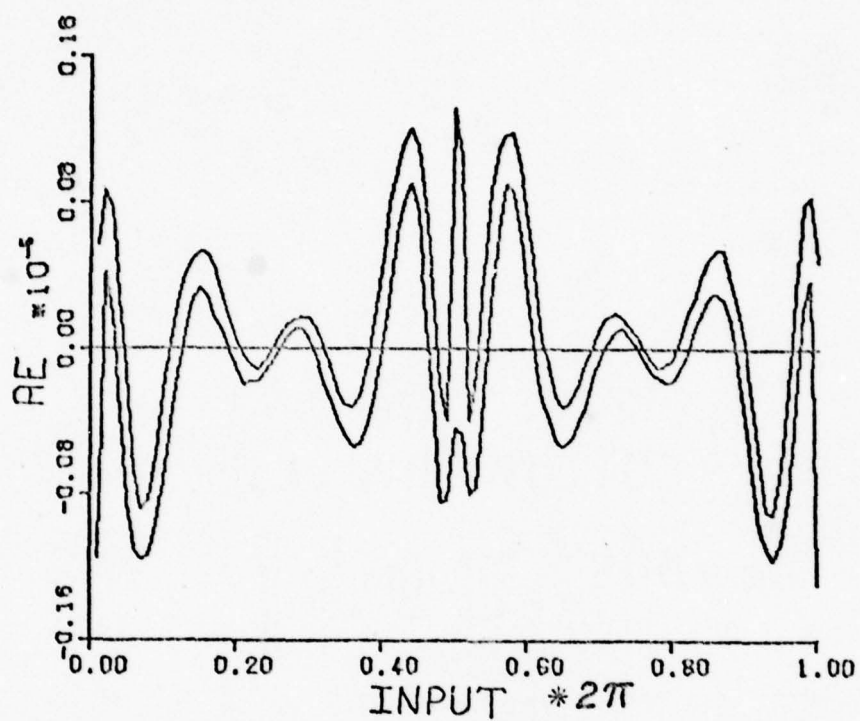


Fig. 21. Absolute Error of Cosine Approximation on $[0, 2\pi]$

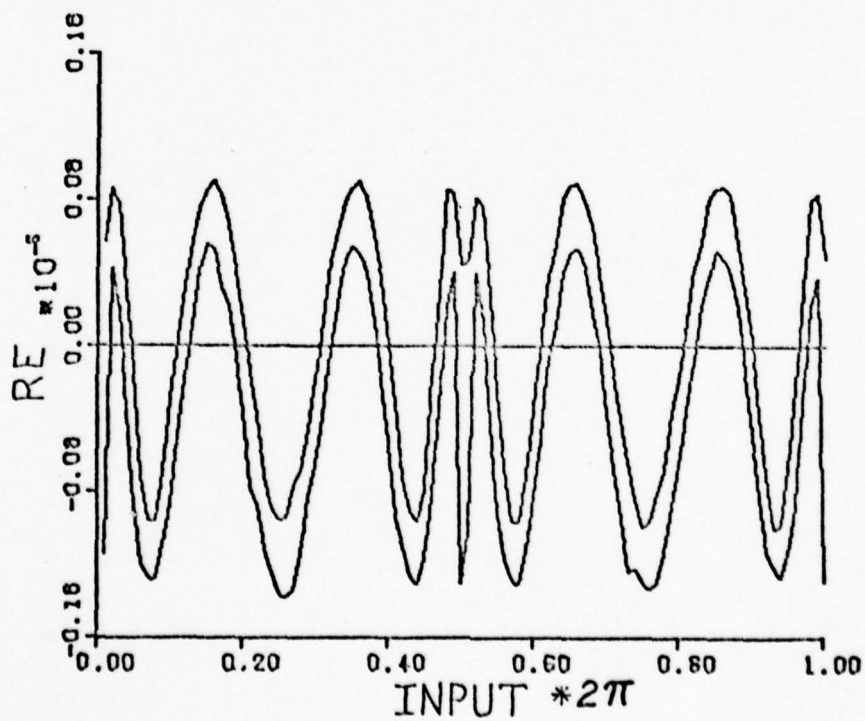


Fig. 22. Relative Error of Cosine Approximation on $[0, 2\pi]$

mirror-image effect being caused by the range reductions.

Summary. When constructing polynomial approximations to the cosine function, range reduction techniques using trigonometric identities are employed. The cosine function can be evaluated using the sine approximation if the identity $\cos(X) = \sin(\pi/2 - X)$ is used. This results in a savings in core storage and also allows the relative error of $\cos(X)$ to be minimized even when $X \approx \pi/2$. Error plots can be used to determine visually the error characteristics of the approximation.

Summary

Common mathematical routines such as sine, cosine, and square root are executed often and need to execute as fast as possible while still providing sufficient accuracy. Range reduction techniques enable lower-order polynomials to be used for approximations, thereby reducing the number of multiplications required. Since cancellation of terms can occur during range reductions, range reductions are performed using extended precision. Minimax polynomials can be used to reduce either the maximum relative or absolute error. Hart, et. al. (Ref 32) gives an extensive collection of minimax and near-minimax solutions to many common functions. Sometimes special applications dictate that error criteria other than the minimax property be used. In these cases, it is usually necessary to know not only the range of the inputs, but also the distribution. Approximations can be tested using Monte Carlo techniques in conjunction with the n-bit simulator. Error plots showing the maximum (positive and negative) errors can be used to aid visually in determining the error characteristics of the approximation.

V Analysis of Avionics Routine

When analyzing the error characteristics of any given processor, it is often helpful to analyze computer programs which would normally be executed on the processor. This is especially important when trying to determine if the wordlength of the processor is long enough to maintain some specified accuracy for the programs, given that the transmitted error has been removed. This chapter contains a detailed discussion of one approach for analyzing the error characteristics of a computer program and the processor on which it is executed. Although only one routine has been used for demonstration purposes, the method presented can easily be applied to other single or multi-variate functions. In order to be analyzed using the method described in this chapter, these functions must be capable of being coded in FORTRAN and executed on either a CDC 6600 or CDC CYBER 74 computer.

A forward error analysis is conducted using the n-bit simulator, and error plots are utilized to aid visually in determining the error characteristics of the routine and the computer on which it would normally be executed. For the purpose of demonstrating this method, the computer being simulated is the AN/AYK-15A digital processor (Ref 2). The AN/AYK-15A uses sine plus two's complement representations of floating point numbers. The processor truncates (as opposed to rounding) and there are no guard bits used.

Bearing To Go Routine

The avionics routine analyzed is the part of the steering function called bearing to go, or desired track (Refs 7 and 57). The steering function is exercised by the AN/AYK-15A during the navigation mode and is used for way-point navigation. Bearing to go (BTG) is the angle in

spherical coordinates between true north and the selected way-point subtended at the present position of the aircraft (using great circle navigation).

BTG is depicted in Fig. 23, where the coordinates (λ, ϕ) represent the present aircraft position and (λ_T, ϕ_T) represent the selected waypoint.

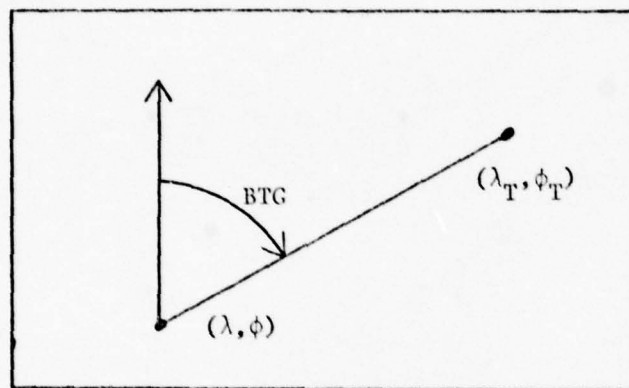


Fig. 23. Bearing To Go

BTG is computed as

$$\text{BTG} = \tan^{-1} \left(\frac{-\text{CT}_y}{\text{CT}_x} \right) \quad (84)$$

where

$$\text{CT}_x = \sin(\lambda_T) * \cos(\lambda) - \cos(\lambda_T) * \sin(\lambda) * \cos(\phi_T - \phi) \quad (85)$$

and

$$\text{CT}_y = -\cos(\lambda_T) * \sin(\phi_T - \phi) \quad (86)$$

λ and λ_T are latitude readings and may take on values in the interval $[-\pi/2, \pi/2]$, while ϕ and ϕ_T are longitude readings and may take on values in the interval $[-\pi, \pi]$. BTG may take on values from the interval $[-\pi, \pi]$ and must have an "accuracy" of 10^{-5} (Ref 57:96). This means that if λ , ϕ , λ_T , and ϕ_T are exactly representable on the AN/AYK-15A (transmitted error removed), then the absolute error (positive or negative) of BTG must not exceed 10^{-5} for any combination of λ , ϕ , λ_T , and ϕ_T .

This routine was chosen for an analysis demonstration for three reasons:

- it is easy to use Monte Carlo techniques to exercise the code, since values for λ , ϕ , λ_T , and ϕ_T can be generated using a pseudo-random number generator (see chapter 3),
- cosine and sine functions are used both in multiplications and additions (or subtractions), so the absolute and relative error characteristics of the sine and cosine approximations must be considered,
- it is a four-variate function, so techniques demonstrated can easily be extended to other multivariate functions.

Bearing To Go Analysis

Objectives. The objectives of this demonstration are to show how the n-bit simulator can be utilized in a forward error analysis and to show how error plots can be utilized to aid visually in analyzing the error characteristics of a multi-variate function. Error plots are used to help show the relationships of each variable to the maximum absolute error. By comparing patterns for different variables, estimates can be obtained for each variable which causes certain large errors to occur. These errors often appear on the plots as prominent spikes.

The n-bit simulator is used in this forward error analysis for two reasons. First, although specifications exist for the AN/AYK-15A digital processor, none have been constructed yet; and second, using the n-bit simulator facilitates comparing results, since data from the simulator and the benchmark data (from the routine executed without the simulator) can be collected at the same time using a driver module.

Approach. The modules used in analyzing the navigation routine BTG

are shown in Fig. 24 and Fig. 25. The modules shown in Fig. 24 are compiled directly (without being preprocessed by the n-bit simulator). The driver module, COMPAR, collects the output from the subroutines SUB1 and SUB2 and computes the absolute and relative errors. COMPAR also develops the data to be used in building the error plots. The range for each variable is divided into 100 evenly-spaced intervals, with each interval containing the maximum absolute and relative errors (both positive and negative). After the simulation has finished, the plot array containing the plot data is written to file TAPE4 to be processed by a plotting program. A functional flow diagram of the driver module COMPAR is shown in Fig. 26.

The subroutine SUB1 shown in Fig. 24 is used to evaluate BTG using the accuracy obtainable with a 48-bit mantissa. The value of BTG returned is used in the forward error analysis as that which is assumed to be without error. Although there is inevitably some error in BTG, it is assumed that it is enough (three orders of magnitude) less than that produced using a 23-bit mantissa that it can be considered to be exact.

The subroutine RANDM does not need to be preprocessed by the n-bit simulator, since it returns values which are exactly representable with a 23-bit mantissa. RANDM calls the CDC pseudo-random generator eight times to get values to use in constructing the exponents and mantissas which make up the four arguments returned. Since RANDM was called many times (20,000), shift operations were inserted to truncate the mantissa to the required accuracy instead of preprocessing RANDM with the n-bit simulator. Therefore, the arrays KEY and TKEY are made available through the COMMON statement.

The modules shown in Fig. 25 are those which are preprocessed by

```

PROGRAM COMPAR(INPUT,OUTPUT,TAPE1=OUTPUT,TAPE4)
COMMON KEY(8),TKEY(4)
CALL SETNBIT(parameters)
:
CALL RANDM( $\lambda$ , $\phi$ , $\lambda_T$ , $\phi_T$ )
    call subroutine to evaluate BTG using full precision
CALL SUB1( $\lambda$ , $\phi$ , $\lambda_T$ , $\phi_T$ ,BTG)
    call subroutine to evaluate BTG using n-bit simulator
CALL SUB2( $\lambda$ , $\phi$ , $\lambda_T$ , $\phi_T$ ,SBTG)
    calculate the absolute and relative errors
AE=SBTG-BTG
RE=0.0
IF (BTG.NE.0.0) RE=AE/BTG
:
STOP
END

SUBROUTINE SUB1( $\lambda$ , $\phi$ , $\lambda_T$ , $\phi_T$ ,BTG)
    subroutine which uses full accuracy of CDC CYBER 74
    calls are made to CDC library for mathematical functions
CTX = SIN( $\lambda_T$ )*COS( $\lambda$ ) - COS( $\lambda_T$ )*SIN( $\lambda$ )*COS( $\phi_T$ - $\phi$ )
CTY = -COS( $\lambda_T$ )*SIN( $\phi_T$ - $\phi$ )
BTG = ATAN2(-CTY,CTX)
RETURN
END

SUBROUTINE RANDM( $\lambda$ , $\phi$ , $\lambda_T$ , $\phi_T$ )
    subroutine to generate pseudo-random inputs
    transmitted errors are removed
COMMON KEY(8),TKEY(4)
:
RETURN
END

```

Fig. 24. Code to be Compiled Without Being Preprocessed

```

SUBROUTINE SUB2 ( $\lambda, \phi, \lambda_T, \phi_T$ , SBTG)
    subroutine which is processed by n-bit simulator
    calls are made to specially coded mathematical functions
COMMON KEY(8),TKEY(4)
SCTX = SSIN( $\lambda_T$ ) * SCOS( $\lambda$ ) - SCOS( $\lambda_T$ ) * SSIN( $\lambda$ ) * SCOS( $\phi_T - \phi$ )
SCTY + - SCOS( $\lambda_T$ ) * SSIN ( $\phi_T - \phi$ )
SBTG = SATAN2(-SCTY,SCTX)
    note - in actual simulation SATAN2 was not used;
    ATAN2 was substituted, and the value returned was truncated
RETURN
END

FUNCTION SSIN(AVALUE)
    specially coded sine approximation (see previous chapter)
COMMON KEY(8),TKEY(4)
    :
ENTRY SCOS
    special entry point for cosine approximation
    :
RETURN
END

FUNCTION SATAN2(AVAL1,AVAL2)
    specially coded inverse tangent approximation
COMMON KEY(8),TKEY(4)
    :
RETURN
END

```

Fig. 25. Code to be N-bit Simulated

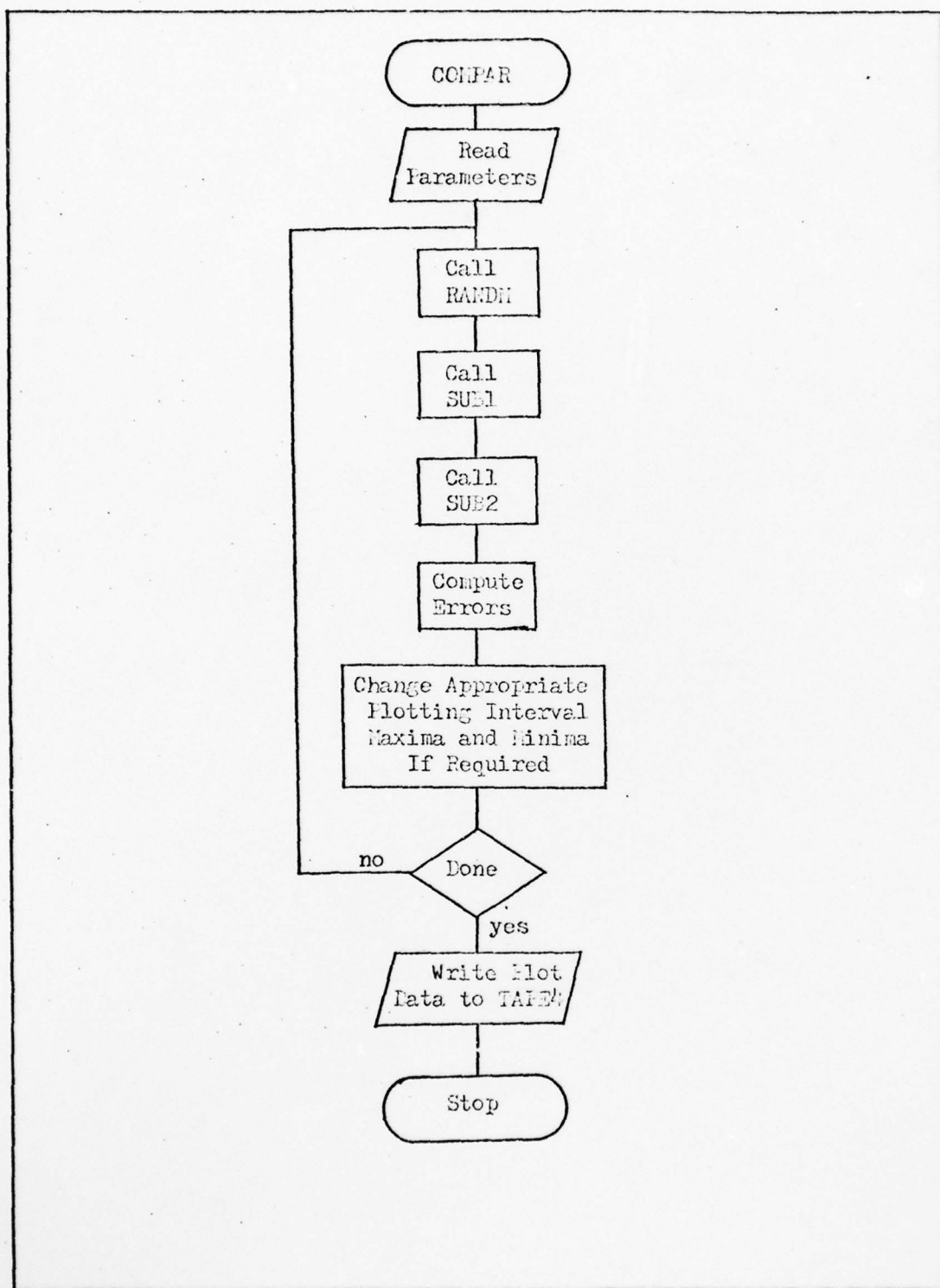


Fig. 26. COMPAR Flow Diagram

the n-bit simulator prior to being compiled. The subroutine SUB2 is used to evaluate BTG using the accuracy which would be obtainable on the AN/AYK-15A processor. The sine function is approximated using the function subprogram SSIN, with an entry point SCOS being used for cosine approximations. The polynomial used in SSIN is the same as that discussed in the previous chapter.

The function subprograms SATAN2 is included in Fig. 25 to show where it would go if it were used. When evaluating the navigation routine BTG the CDC library routine ATAN2 was used, with the mantissa of the result being truncated to the number of bits specified (23 bits). This was because the arctangent function was not analyzed as the sine and cosine functions were.

Termination Criteria. The procedure used for computing n, the number of trials to conduct, follows that discussed in chapter 3. The equation

$$n \geq \frac{\ln(\beta)}{\ln(1-P)} \quad (50)$$

is used to solve for n once β and P have been determined. β , which is the tester's risk that he accepts bad software, was arbitrarily specified to be 0.01. By decreasing β , the tester becomes more confident that software he accepts actually meets the specifications. However, decreasing β also increases the computed value of n, the number of trials he must conduct. To determine P, the equation

$$P = \sum_{i=1}^V \frac{p_i}{w_i} - \sum_{\substack{i,j \\ i \neq j}} \frac{p_i}{w_i} \cdot \frac{p_j}{w_j} + 0^3 \quad (49)$$

is used with all terms of order three or greater being truncated. V, the number of variables, is four. The four variables p_i are assumed to

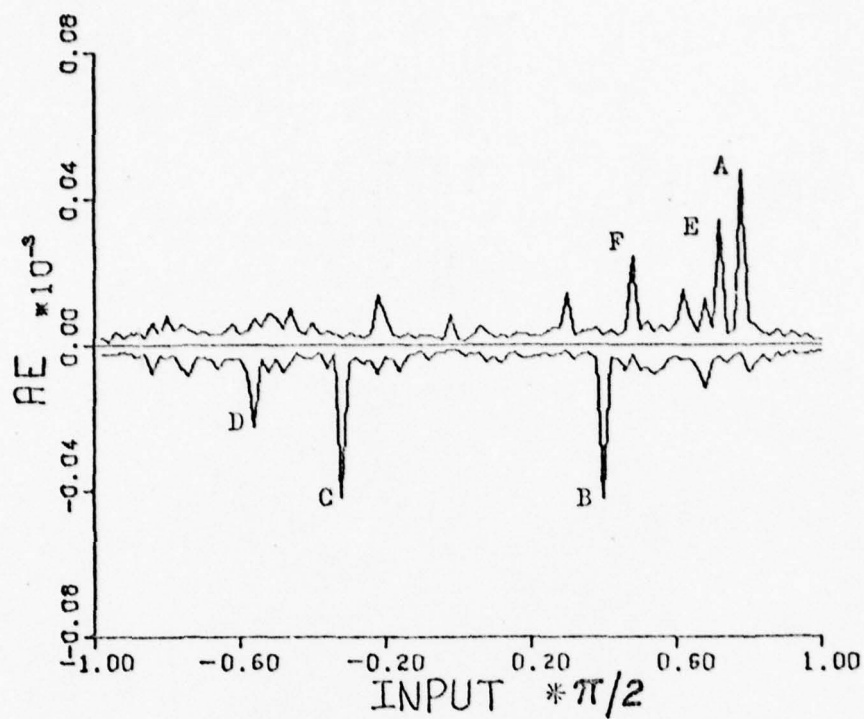


Fig. 27. Absolute Error vs BTG Input Variable λ

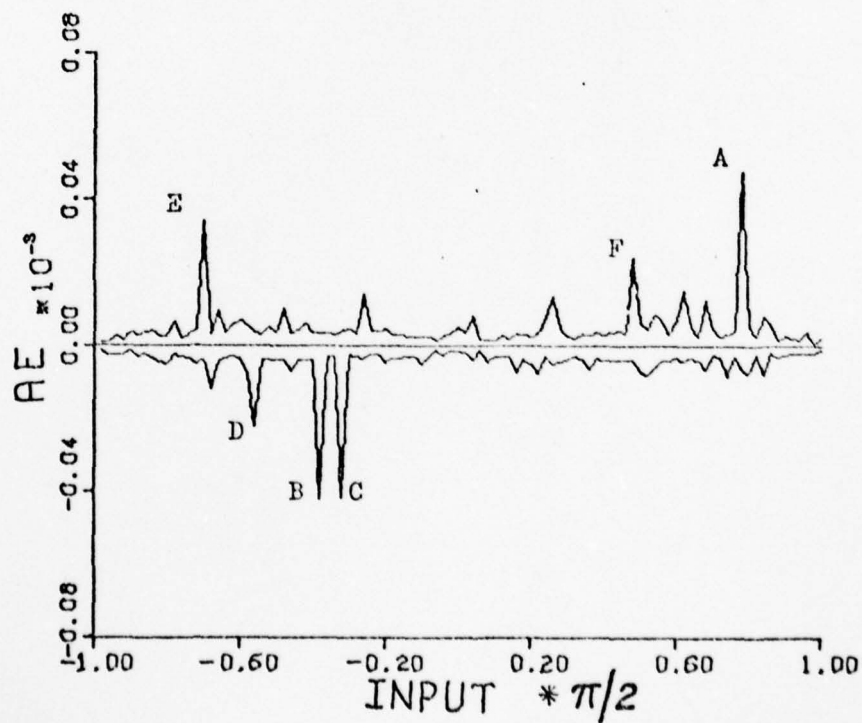


Fig. 28. Absolute Error vs BTG Input Variable λ_T

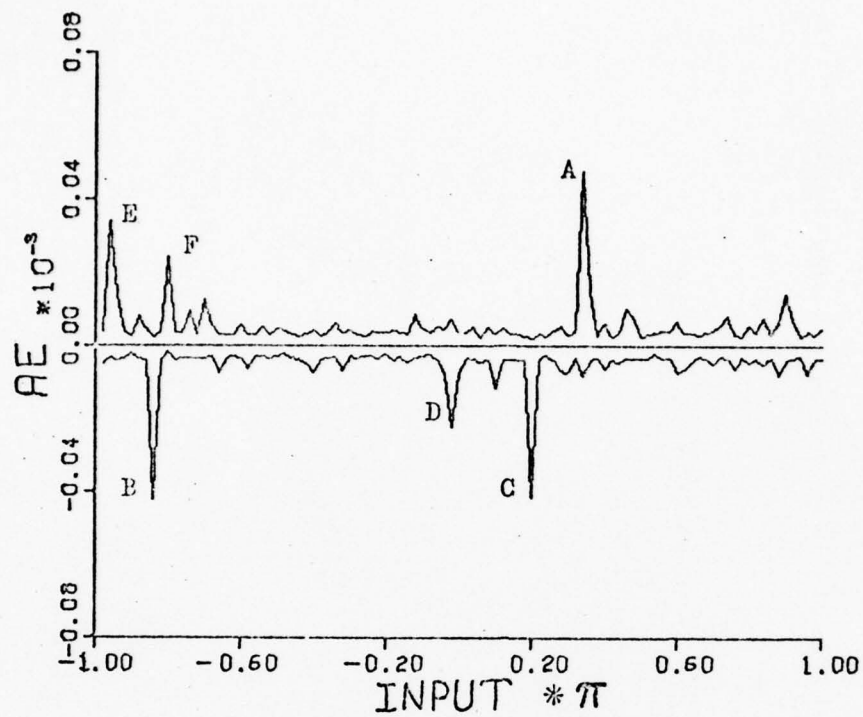


Fig. 29. Absolute Error vs BTG Input Variable ϕ

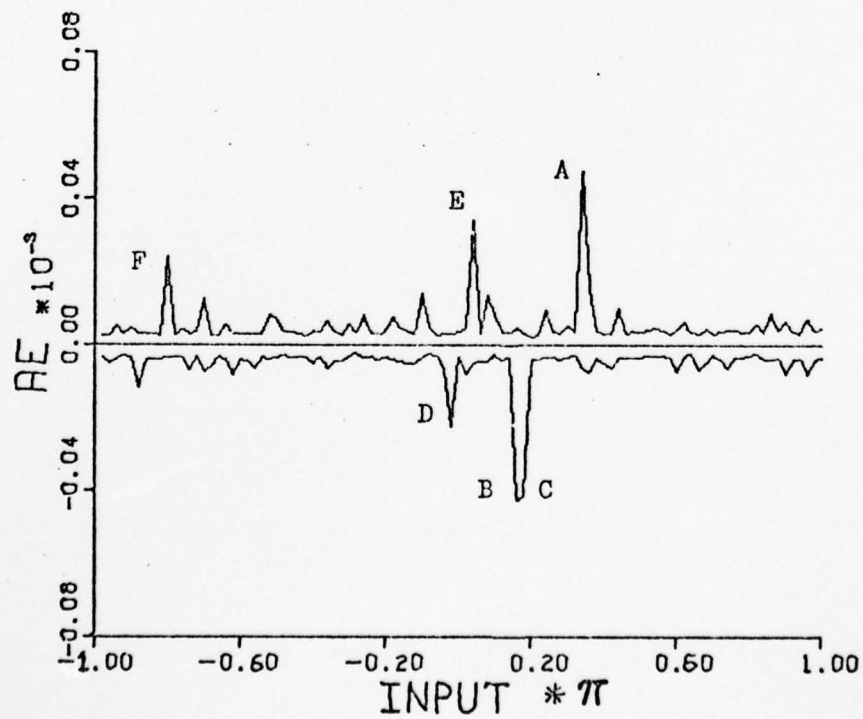


Fig. 30. Absolute Error vs BTG Input Variable ϕ_T

the maxima obtained. These plots are shown in Figs. 31, 32, 33, and 34. As can be seen, the absolute error maximum (either positive or negative) is much less than that obtained when the 20,000 bit patterns were used in conjunction with these 5000 numbers. Realizing that the CDC library function ATAN2 was substituted for an n-bit approximation for the arc-tangent function, one might be tempted to accept the BTG function as meeting the accuracy specifications if only the errorplots using the 5000 trials were considered. However, it can be seen by looking at the plots constructed using the 25,000 trials that the maximum error obtained is at least five times larger, and that the BTG routine, if coded in the same manner as tested, would fail to meet the specifications. For the duration of this chapter, reference to error plots will mean those constructed using the 25,000 trial numbers (Figs. 27, 28, 29, and 30).

Once the routine, as coded, has been determined to fail to meet the accuracy specifications, it remains to be determined whether a judicious usage of extended precision might help. As can be seen from the error plots, most of the values tested fall within the specified limit of 10^{-5} . However, several large spikes exceed the limit by a factor of 2 and several smaller spikes also exceed the limit. Error values of 0.005×10^{-3} , or 0.5×10^{-5} , occur very regularly and are to be expected since there is some error generated within the sine and cosine routines. The large spikes, however, are not expected. Each spike which will be analyzed is marked by a letter on each of the plots. It is only conjectured that the spikes are related on each of the four plots by the letters assigned them. Although spikes A and B can be clearly identified, since they are the absolute maxima (positive and negative), the others cannot be so positively identified without more error trace information provided

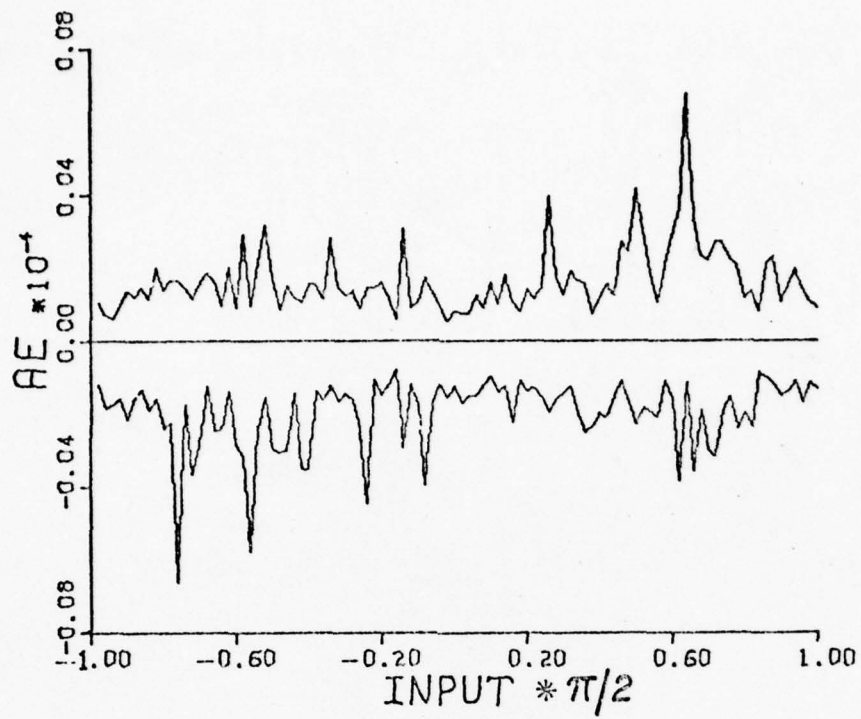


Fig. 31. Absolute Error Using 5000 Numbers vs λ

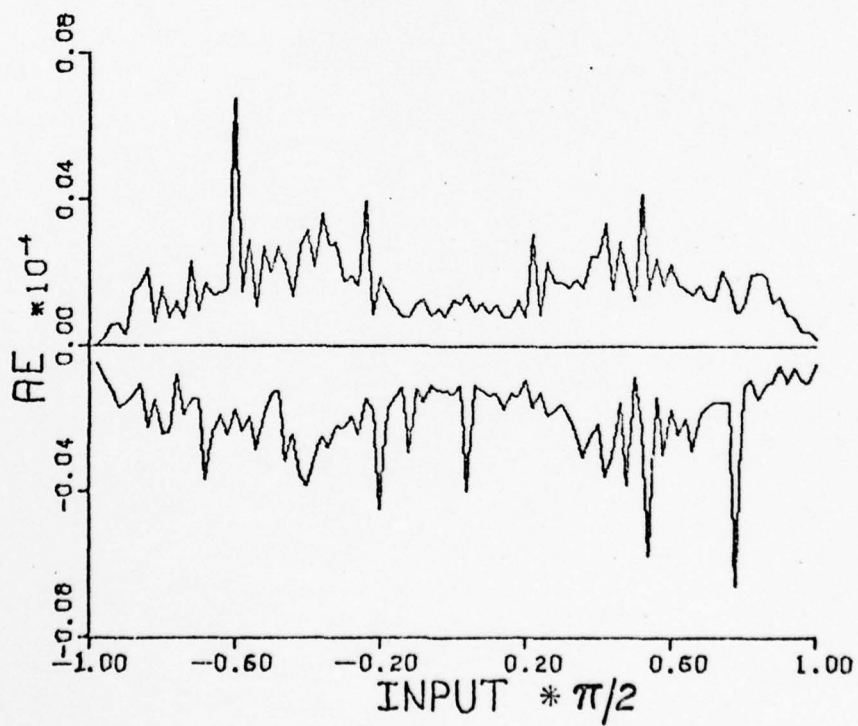


Fig. 32. Absolute Error Using 5000 Numbers vs λ_T

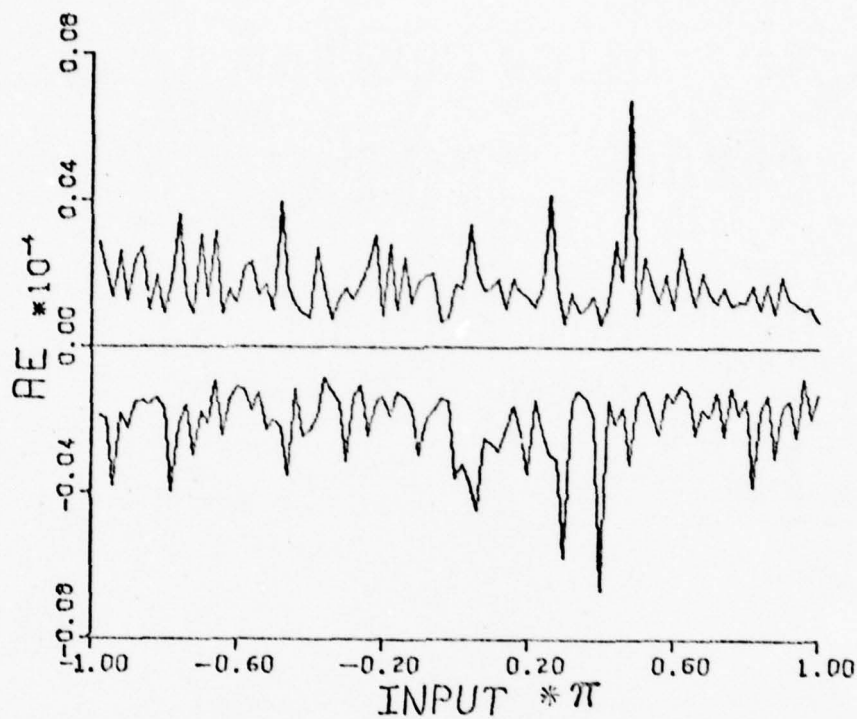


Fig. 33. Absolute Error Using 5000 Numbers vs ϕ

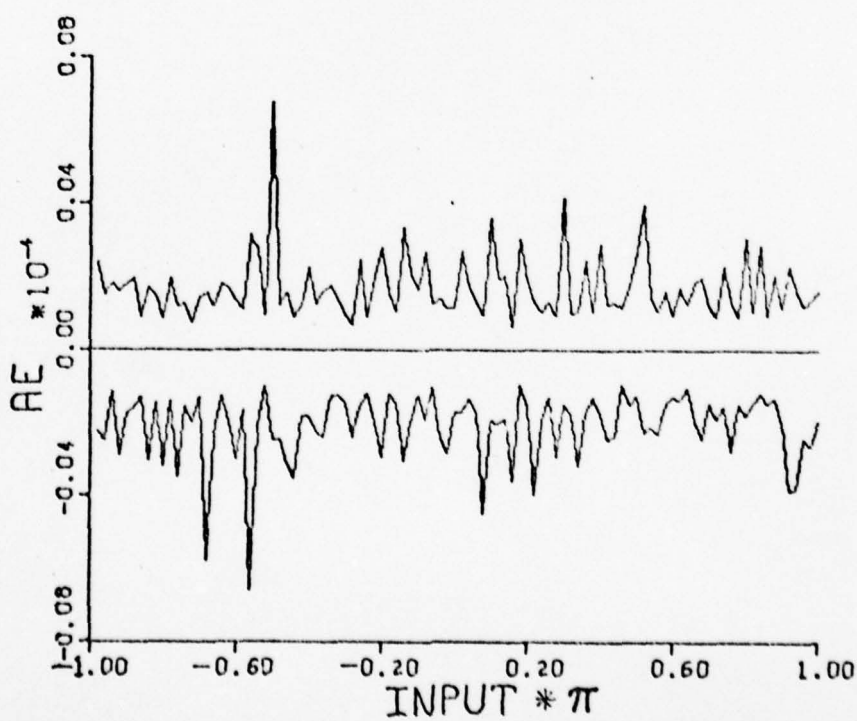


Fig. 34. Absolute Error Using 5000 Numbers vs ϕ_T

by the driver module COMPAR. However, for the purposes of this analysis, it is assumed that spikes A through F can be identified on each plot and that the tip of each spike with the same letter identifier is caused by the same set of trial numbers. This allows each spike to be analyzed independently of the other five identified. In Fig. 30, spikes B and C occur in neighboring plotting intervals, with spike B occurring to the left of spike C.

Spike Analysis. Since the BTG function has subtractions, it might be hypothesized that the spikes are caused by cancellation of terms. If so, then the values of the input variables which cause the spikes should clearly show this once extracted from the plots. Each of the six spikes identified will be analyzed separately.

Spike A. Spike A is the maximum positive error discovered, so there is no problem identifying the values of λ , λ_T , ϕ , and ϕ_T which cause spike A to occur. If the plots for λ and λ_T are compared, it can be seen that the spike actually occurs on the same plotting interval. This says that $\lambda \approx \lambda_T$. Likewise, the plots of ϕ and ϕ_T can be compared with the like conclusion that $\phi \approx \phi_T$. This says that the aircraft position (λ, ϕ) and the waypoint location (λ_T, ϕ_T) are approximately the same place. Since $\phi_T \approx \phi$, $\sin(\phi_T - \phi) \approx 0$, and hence $CT_y \approx 0$. $\cos(\phi_T - \phi) \approx 1$, so CT_x reduces (approximately) to $CT_x \approx \sin(\lambda_T) \cos(\lambda) - \cos(\lambda_T) \sin(\lambda)(1)$. Now, CT_x has taken the form of $\sin(\lambda_T - \lambda)$, which is clearly zero. Therefore, the ratios $-CT_y/CT_x$ tends toward 0/0, and $\tan^{-1}(-CT_y/CT_x)$ can be expected to exhibit large errors.

Spike B. For spike B, $\lambda \neq \lambda_T$, and $\phi \neq \phi_T$, so each must be approximated to determine why the spike occurred. As can be seen on the plots, $\lambda \approx \pi/5$, $\lambda_T \approx -\pi/5$, $\phi \approx \frac{-22\pi}{25}$, and $\phi_T \approx \frac{3}{25}\pi$. For these values $\phi_T - \phi \approx +\pi$, and \sin

$(\phi_T = \phi) \approx 0$, so $CY_y \approx 0$. Likewise, $\cos(\phi_T - \phi) \approx -1$, and CT_x is reduced (approximately) to

$$CT_x \approx \sin(\lambda_T) \cos(\lambda) - \cos(\lambda_T) \sin(\lambda)(-1)$$

CT_x now has the form of $\sin(\lambda_T + \lambda)$, which also goes to zero since $\lambda_T \approx -\lambda$. Thus, spike B is also caused by the instability of the argument $-CY_y/CT_x$ approaching 0/0.

Spike C. Spike C exhibits the same characteristics as spike A, where $\lambda \approx \lambda_T$ and $\phi \approx \phi_T$.

Spike D. Spike D exhibits the same characteristics as spike A, where $\lambda \approx \lambda_T$ and $\phi \approx \phi_T$.

Spike E. Spike E exhibits the same characteristics as spike B, with $\lambda \approx -\lambda_T$ and $\phi \approx \phi_T - \pi$. Therefore, $CT_y \approx 0$ and CT_x approaches the form $\sin(\lambda_T + \lambda)$ as $(\phi_T - \phi)$ approaches π . Since $CT_x \approx 0$, the argument $-CT_y/CT_x$ approaches 0/0.

Spike F. Spike F exhibits the same characteristics as spike A, where $\lambda \approx \lambda_T$ and $\phi \approx \phi_T$.

Recommendations

Those conditions which cause spikes A and B to occur are special cases which cannot be expected to occur very often in aircraft navigation. The conditions which cause spike A to occur ($\lambda \approx \lambda_T$ and $\phi \approx \phi_T$) are similar to the problem of trying to use a magnetic compass when standing on the magnetic north pole, and the conditions which cause spike B to occur ($\lambda \approx -\lambda_T$ and $\phi \approx \phi_T - \pi$) are similar to the problem of finding the shortest route to the true south pole while standing on the true north pole. However, these conditions are discovered when testing with pseudo-random trial numbers.

To preclude these special occurrences from affecting the simulated

results, two arbitrary "circles" with radii r_1 and r_2 should be constructed around the point (λ, ϕ) . The first circle, with radius r_1 , is used to detect the condition when the aircraft location and the waypoint are approximately the same. This condition is detected by requiring that the great circle distance between (λ, ϕ) and (λ_T, ϕ_T) be greater than r_1 . The second circle is used to detect the condition when the waypoint is on the opposite side of the earth from the aircraft. This condition is detected by requiring that the distance between (λ, ϕ) and (λ_T, ϕ_T) be less than r_2 . Once these two conditions have been imposed on λ_T and ϕ_T (for a given λ and ϕ), simulations can be conducted which more accurately reflect what might actually occur in an aircraft environment.

To compute the distance (DIS) that (λ_T, ϕ_T) is from (λ, ϕ) , the spherical distance equation

$$DIS = (R+h) * \sigma_T \quad (87)$$

may be used. R is an approximation ($2.0926 * 10^7$ feet) for the radius of the earth, h is the aircraft altitude in feet, and σ_T may be computed by

$$\sigma_T = \tan^{-1} \frac{CT_x * \cos(BTG) - CT_y * \sin(BTG)}{CT_z} \quad (88)$$

BTG , CT_x , and CT_y are computed as shown in equations (84), (85), and (86) respectively, and CT_z is computed by

$$CT_z = \sin(\lambda) * \sin(\lambda_T) + \cos(\lambda) * \cos(\lambda_T) * \cos(\phi_T - \phi) \quad (89)$$

Once DIS is computed, then those values of λ , ϕ , λ_T , ϕ_T for which $DIS < r_1$ or $DIS > r_2$ may be discarded. Since BTG must be computed to get DIS , different error plots can easily be generated using different values of r_1 and r_2 .

It is conjectured that the BTG routine would actually meet the

accuracy specifications once distance constraints have been levied against λ_T and ϕ_T . This conjecture is based solely on the width of the general small error band centered about zero and was not actually tested. In order to retest the BTG routine incorporating the distance constraints, those cases where (λ, ϕ) and (λ_T, ϕ_T) fail to meet the distance criteria must be discarded. One method for testing using the distance constraints would be to pick a value of (λ, ϕ) , and then generate (λ_T, ϕ_T) coordinates until a set of coordinates is found which meets the distance criteria. If it is assumed that the proportion of the earth's surface which meets the distance criteria for any given point is E, then N, the number of points (λ_T, ϕ_T) which one might expect to generate for each point (λ, ϕ) , may be computed by

$$N = 1/(2E) \quad (99)$$

To test the BTG routine, n points (λ, ϕ) would be generated, where n is computed using equation (50). Also, $n/(2E)$ points (λ_T, ϕ_T) would be generated, with the distance formula being computed for each one. Since the purpose of this investigation was to demonstrate a technique for analyzing flight routines as opposed to actually conducting a thorough analysis, the routine BTG was not tested using the distance criteria.

Summary

When conducting a forward error analysis utilizing the n-bit simulator, a driver module can be used to facilitate error data gathering. Two subroutines can be used, with one being executed using the full accuracy obtainable on the CDC CYBER 74 and the other being reprocessed along with any mathematical function approximations needed. The object decks for the n-bit preprocessed subroutine and mathematical functions, the n-bit subroutines (see chapter 2), and the driver module and regular

function are combined at load time before execution.

Error characteristics can be studied using plots which show the error maxima (positive and negative). When simulating, it becomes important to consider only those values of variables which can actually occur in an operational environment. When values are arbitrarily chosen from over the entire range of each variable, errors can sometimes occur which affect the results of the simulation. For this reason, additional constraints must be placed on the pseudo-random inputs used. The use of error plots can sometimes point out these situations, since relationships which exist between variables can also be observed in addition to the maximum errors.

VI Quasilinearization Method

When numerical software routines are analyzed by Monte Carlo techniques, the tester is faced with determining the number of random samples to use to ensure that his risk is less than some desired value. No matter what technique is employed to compute the number of samples required, the tester is faced with making assumptions about the error characteristics of the software.

For any set of input values for a software routine, the error (either relative or absolute) can be uniquely determined given the characteristics of the executing computer. In the following discussion the term error is considered to mean the absolute error in a function. Relative errors could also be used, since they are obtained by dividing the absolute error by the true value of the function (provided the true value of the function is not zero). Since the error is uniquely determined, it can be considered to be a discrete function of the implementation of some algorithm (A), the characteristics of the machine (M), and the independent variables (v_i).

$$ER = f(A, M, v_1, v_2, \dots, v_n) \quad (91)$$

For the BTG routine, the variables v_i , $i = 1, 2, 3, 4$, would represent the input variables λ, ϕ, λ_T , and ϕ_T . The variables v_i can be considered to form a vector Q, and for the purposes of this discussion, A and M represent a given algorithm implemented on a given computer. Since A and M are specified, ER will be written as $f(Q)$. The points $(Q_i, f(Q_i))$ are considered to be a subset of an $(n+1)$ - dimensional vector space.

The quasilinearization method (Ref 67) as proposed in this chapter is a suboptimal search technique for finding local maxima of the function

AD-A069 299

AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OHIO SCH--ETC F/G 9/2
IMPLEMENTATION AND TESTING OF NUMERICAL ANALYSIS TECHNIQUES IN --ETC(U)
MAR 79 R A ADAMS

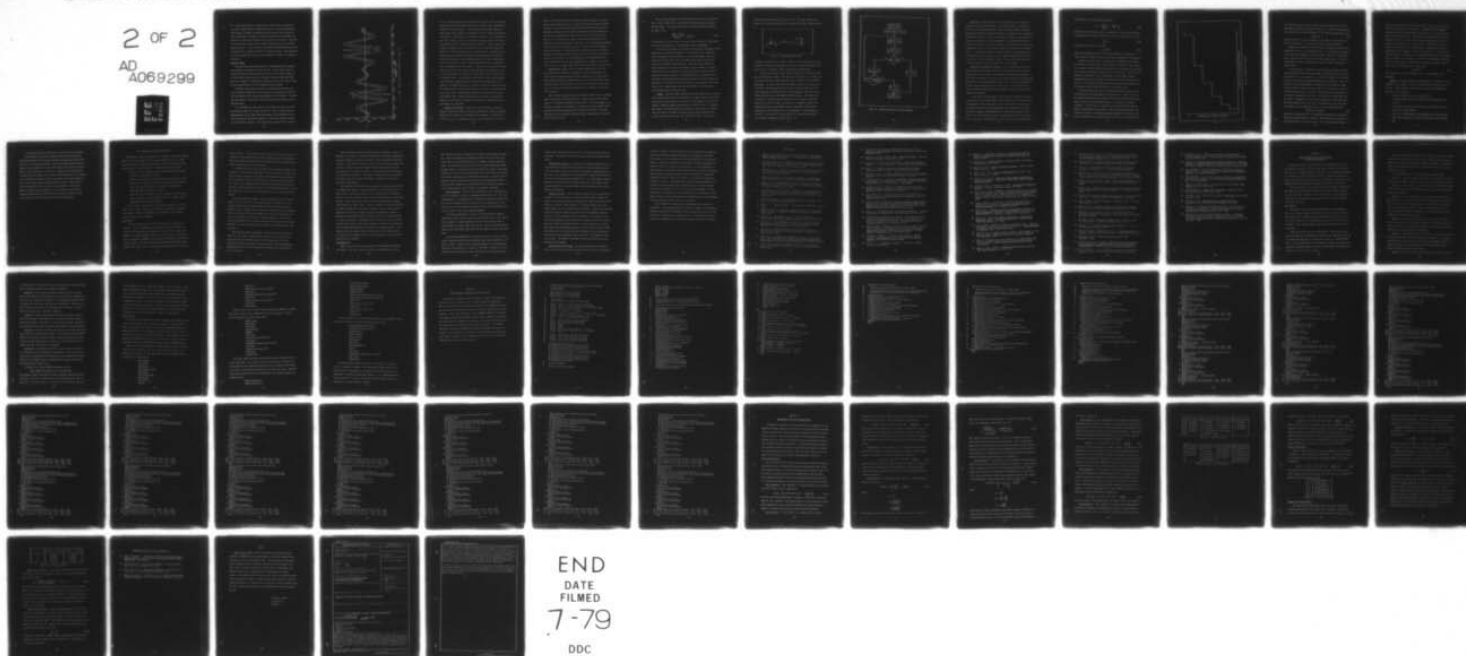
UNCLASSIFIED

AFIT/GCS/EE/79-1

NL

2 OF 2

AD
A069299



END
DATE
FILMED
7-79
DDC

ER. Local maxima which are discovered are entered in a population to be used in determining the stopping criterion. In the quasilinearization method, the number of individual tests required is not based on some hypothesized percentage of successes caused by each variable, but rather on a statistical analysis of the results of each test. Each test is a more lengthy process than a simple evaluation of a set of random inputs, however. In this chapter, the general quasilinearization method is discussed as it might be applied to software testing. Also discussed is a way in which test results can be analyzed to arrive at a stopping criterion.

Proposed Method

The quasilinearization method uses a suboptimum gradient approach in an attempt to extremize the function ER. If the tester knows the extremum of ER has been obtained, he can say with certainty whether or not the routine being tested ever exceeds the error bounds specified. Since this method is a suboptimal procedure, however, the tester seldom, if ever, knows when or if the extremum has been obtained.

Gradients, when used in calculus to extremize multivariate functions, generally required that the function be continuous and that the first derivative exist (once-differentiable) (Ref 8:349-404). The function ER, however, is strictly a discrete-valued function which is non-differentiable.

It is assumed that the values of the function ER often lie within well-defined limits as is shown in Fig. 35. The two lines represent approximations of the upper and lower bounds to the absolute error of the seventh-order minimax approximation to the sine function executed on the AN/AYK-15A computer. The upper bound as drawn will be referred

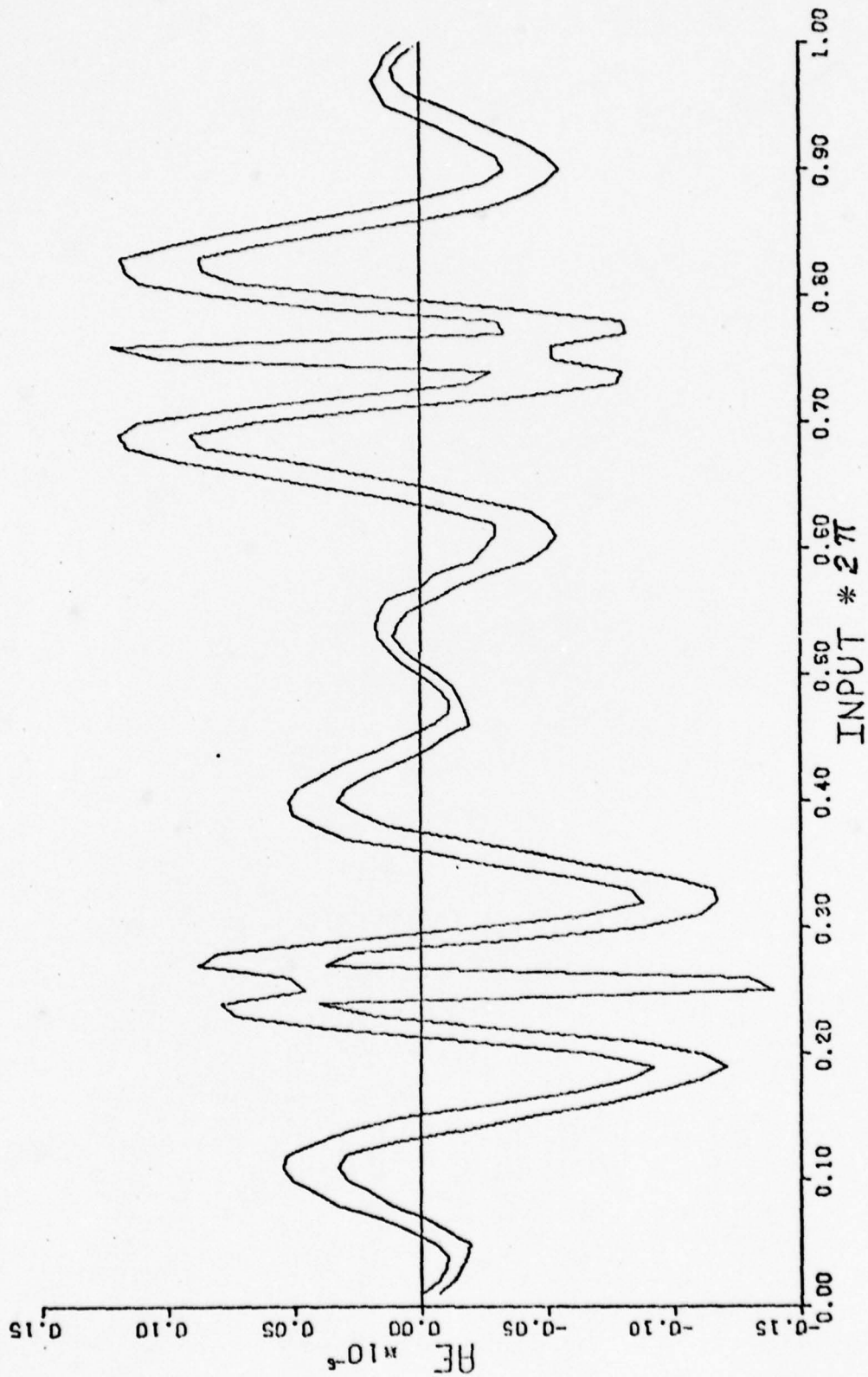


FIG. 35. Absolute Error Bound for Sine (x) on $[0, 2\pi]$

to as a continuous and once differentiable function, since a continuous and once-differentiable upper bound can be shown to exist (although not necessarily the one drawn). If the equation of the upper bound were known, then vector analysis and gradients could be applied to extremize the function. This equation is not known, however. It is assumed that all that is known or can be discovered are various values of ER which lie somewhere between the upper and lower bounds. For two values of ER ($f(Q_1)$ and $f(Q_2)$) which lie arbitrarily close to the upper bound and which are a "small" distance apart, a straight line, or linear spline, connecting the two points $(Q_1, f(Q_1))$ and $(Q_2, f(Q_2))$ can be obtained. This spline provides a linear approximation to the upper bound and the slope of the spline is used as a directional difference approximation for the gradients at the two points. Two values Q_1 and Q_2 can be found (with some difficulty) for which $f(Q_1)$ and $f(Q_2)$ lie arbitrarily close to the upper bound. The method of obtaining the values Q_1 and Q_2 will be explained later. By using the spline and its slope, normal methods for extremizing continuous and once-differentiable functions can be applied. The method discussed is a linear exploration in which searches are conducted only in directions parallel to one axis and perpendicular to all other axes. Several other methods are discussed by Beveridge and Schechter (Ref 4) and by Wilde (Ref 64). Before the algorithm is presented, the concepts of moving to the surface, obtaining a spline, and jumping are discussed.

Moving to the Surface. The process of determining a point $(Q_j, f(Q_j))$ for which the function $f(Q_j)$ lies arbitrarily close to the upper bound will be referred to as "moving to the surface". For any set of variables v_1, v_2, \dots, v_n making up Q_i , the error function ER, or $f(Q_i)$ is

known to lie somewhere between the upper and lower bounds. Each variable is then perturbed, one at a time, by the smallest amount possible m_j times in both the positive and negative directions (with the other variables remaining fixed) and ER is evaluated for each perturbation. The smallest amount possible means that the new value is the number which, on the floating-point number line for the given machine, lies adjacent to the original number. The value of m_j is arbitrary depending on the effort one is willing to expend. As more effort is expended, more perturbations are evaluated, thus raising the tester's confidence that he has obtained a point for which the error is within a specified distance of the surface. The maximum value of ER obtained from all these evaluations (at some point $(Q_j, f(Q_j))$) is assumed to lie sufficiently close to the upper bound, and one has thus "moved to the surface".

Obtaining the Spline. One of the most difficult parts of the quasilinearization method is to find two values Q_1 and Q_2 which are close together (using Euclidean distance measures for vectors) and for which $f(Q_1)$ and $f(Q_2)$ lie arbitrarily close to the upper bound, since the upper bound is never known. The success of the process described here is related to the effort expended, just as in the case of moving to the surface.

Once a point $(Q_1, f(Q_1))$ has been obtained, another point $(Q_2, f(Q_2))$ must be obtained in order to construct a linear spline, since linear splines are uniquely determined by two points. To get $(Q_2, f(Q_2))$, the process of moving to the surface is repeated at point $(Q_1, f(Q_1))$, with m_2 ($j=2$) perturbations being used in each axis direction. The point $(Q_2, f(Q_2))$ must be different from $(Q_1, f(Q_1))$ and $f(Q_2)$ must be greater than or equal to $f(Q_1)$.

If a point $(Q_2, f(Q_2))$ can be found which satisfies these conditions, then a linear spline can be constructed between the points $(Q_1, f(Q_1))$ and $(Q_2, f(Q_2))$. If $d(Q_1, Q_2)$ represents the Euclidean distance between Q_1 and Q_2 , then

$$\frac{f(Q_2) - f(Q_1)}{d(Q_2, Q_1)} = g(Q_1, Q_2) \quad (92)$$

is considered to be the slope of the spline and is used as a directional difference approximation to the gradient at point $(Q_1, f(Q_1))$.

If no point $(Q_2, f(Q_2))$ can be found, then the point $(Q_1, f(Q_1))$ is not assumed to be a true local maximum. The reason for this is that the point $(Q_1, f(Q_1))$ may not actually be a true local maximum, since the existence of sharp ridge lines can produce false maxima using linear exploration techniques (Ref 64:65-68). For this reason, whenever a local maximum has been discovered (whether true or false), the linear exploration can be expanded to search along directions not previously searched using axis rotations. Although this does not guarantee that the final local maximum obtained is not a false one, it does provide an improvement to the basic linear exploration in that problems with ridge lines will be reduced (but still not eliminated). The treatment of local maxima will be discussed later.

Jumping. Once two points $(Q_1, f(Q_1))$ and $(Q_2, f(Q_2))$ have been found and the linear spline has been constructed, a jump is made to find a new point which, after moving to the surface to $(Q_3, f(Q_3))$, may or may not produce a monotone nondecreasing sequence of values of ER. The jump is made based on the differences in the values of one of the v_i variables (the one which is different between Q_1 and Q_2) and the slope of the spline (the steeper the slope, the smaller the jump). This jump is

shown for a two-dimensional case in Fig. 36. The value "jumped to" is shown as Y , and Q_3 is obtained after moving to the surface from location Y .

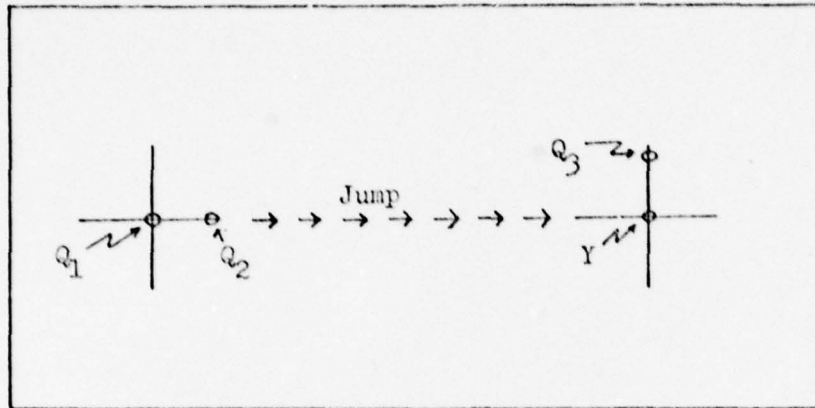


Fig. 36. Quasilinearization Jump

If $f(Q_3)$ is greater than or equal to $f(Q_2)$, then the jump has been successful, and the point $(Q_3, f(Q_3))$ becomes the new point $(Q_1, f(Q_1))$, with the process of obtaining a spline and jumping starting again.

If $f(Q_3)$ is less than $f(Q_2)$, then the jump has not been successful and a new jump must be made. The length of the jump is cut in half, and the jump is repeated, followed by the move to the surface to establish $(Q_3, f(Q_3))$. If, after some arbitrary number of jumps, no value Q_3 can be found for which $f(Q_3)$ is greater than or equal to $f(Q_2)$, then the point $(Q_2, f(Q_2))$ can become the new point $(Q_1, f(Q_1))$, with the process of obtaining the spline and jumping starting again. This assures the existence of a monotone nondecreasing sequence of points $(Q_1, f(Q_1))$ based on the value of $f(Q_1)$. The process of obtaining a spline and jumping terminates when a spline cannot be obtained, and the point $(Q_1, f(Q_1))$ is then assumed to be a local maximum. As mentioned previously, it is possible for $(Q_1, f(Q_1))$ to be a false local maximum. A flow diagram for jumping is shown in Fig. 37.

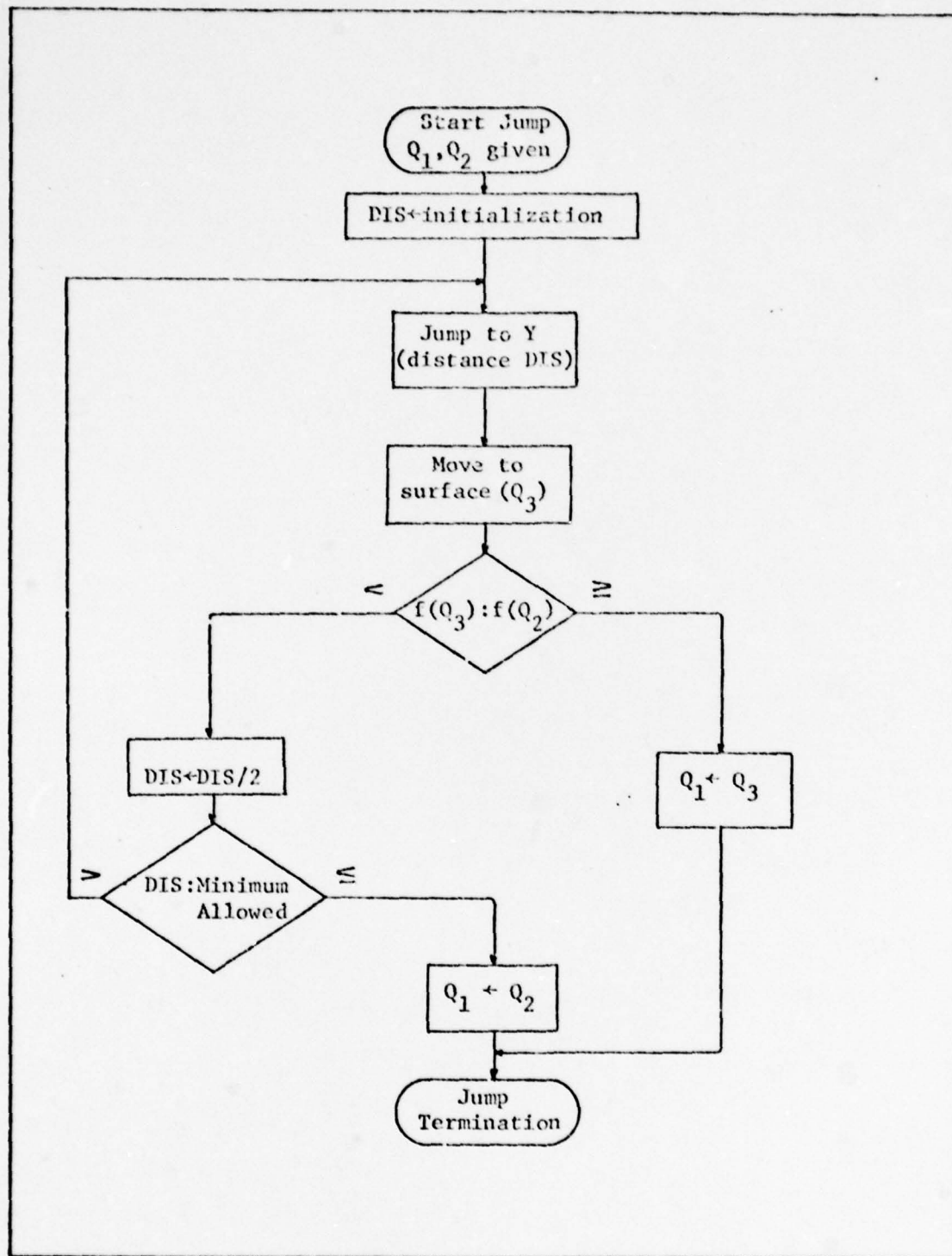


Fig. 37. Quasilinearization Jump Flow Diagram

Algorithm. Each test consists of using Monte Carlo techniques to obtain an initial starting point. Each input variable v_i comprising the vector Q is generated using the entire variable range to construct uniformly-distributed pseudo-random values, as opposed to the techniques discussed in chapter 3. From this point Q , the process of moving to the surface, obtaining the spline, and jumping is repeated until a local maximum is obtained. This local maximum is then entered as one value in a population which is statistically analyzed to determine the stopping point. As each local maximum is entered in order in the population, it is identified as being either an "old" sample, meaning that that particular value (or one arbitrarily close to it) was previously obtained as a local maximum, or else it is entered as a "new" sample.

Analysis of Population of Maxima. It is assumed that there are N local maxima which can theoretically be found and each has an equally likely chance of occurring. The problem of analyzing the population of "new" and "old" samples to estimate the stopping point for the algorithm thus becomes one of estimating the value of N . It is also assumed that, of the N local maxima which can be found, one is larger than the rest and is therefore identified as the extremum of the function ER over the ranges of the inputs.

If the tester has not found a success (value of $f(Q)$ which exceeds some specified error bound) after z "new" samples have been entered in the population, the tester would at least like to have a confidence of $1-\alpha$ that the largest local maximum found is the extremum. For each of the N local maxima, the probability of it being the extremum is $1/N$. After z new local maxima have been discovered, there are $N-z$ remaining to be discovered. The probability ξ that none of the $N-z$ undiscovered

local maxima is the extremum is therefore

$$\xi = 1 - \left(\prod_{i=1}^{N-z_1} \frac{N}{N} \right) = 1 - \frac{N-z}{N} = \frac{z}{N} \quad (93)$$

Because the tester wishes to have a confidence of at least $1-\alpha$ that the extremum has already been encountered, ξ must be greater than $1-\alpha$; i.e.

$$\frac{z}{N} \geq 1-\alpha \quad (94)$$

Solving for z in terms of N yields

$$z \geq N(1-\alpha) \quad (95)$$

Thus, the problem becomes one of estimating N given the sequence of "old" and "new" samples entered in the sample population of local maxima.

Whenever a "new" sample is entered in the population, the cumulative "time" to that point (one time quantum = one trial) is also entered. It is assumed that the sample points come from a continuous distribution, even though the cumulative number of trials is discrete. This allows an empirical distribution to be constructed. For the sample sequence nnonoonooonooooooooon, the empirical distribution looks like that shown in Fig. 38. If the type of distribution and the associated parameters can be determined, then \hat{N} can be obtained as an estimate for N .

When determining the type of distribution which best fits the sample data points, many distributions should be tried. "Best" fits can be obtained using either the Legendre least squares method, the Gregory-Newton method, the mean average method, or the method of weighted residuals. In each case, the parameters of the distributions must be obtained analytically from the data. Once the distributions have been fitted to the data, the Kolmogorov-Smirnov goodness-of-fit test can be used to eliminate those distributions which do not fit the sample data.

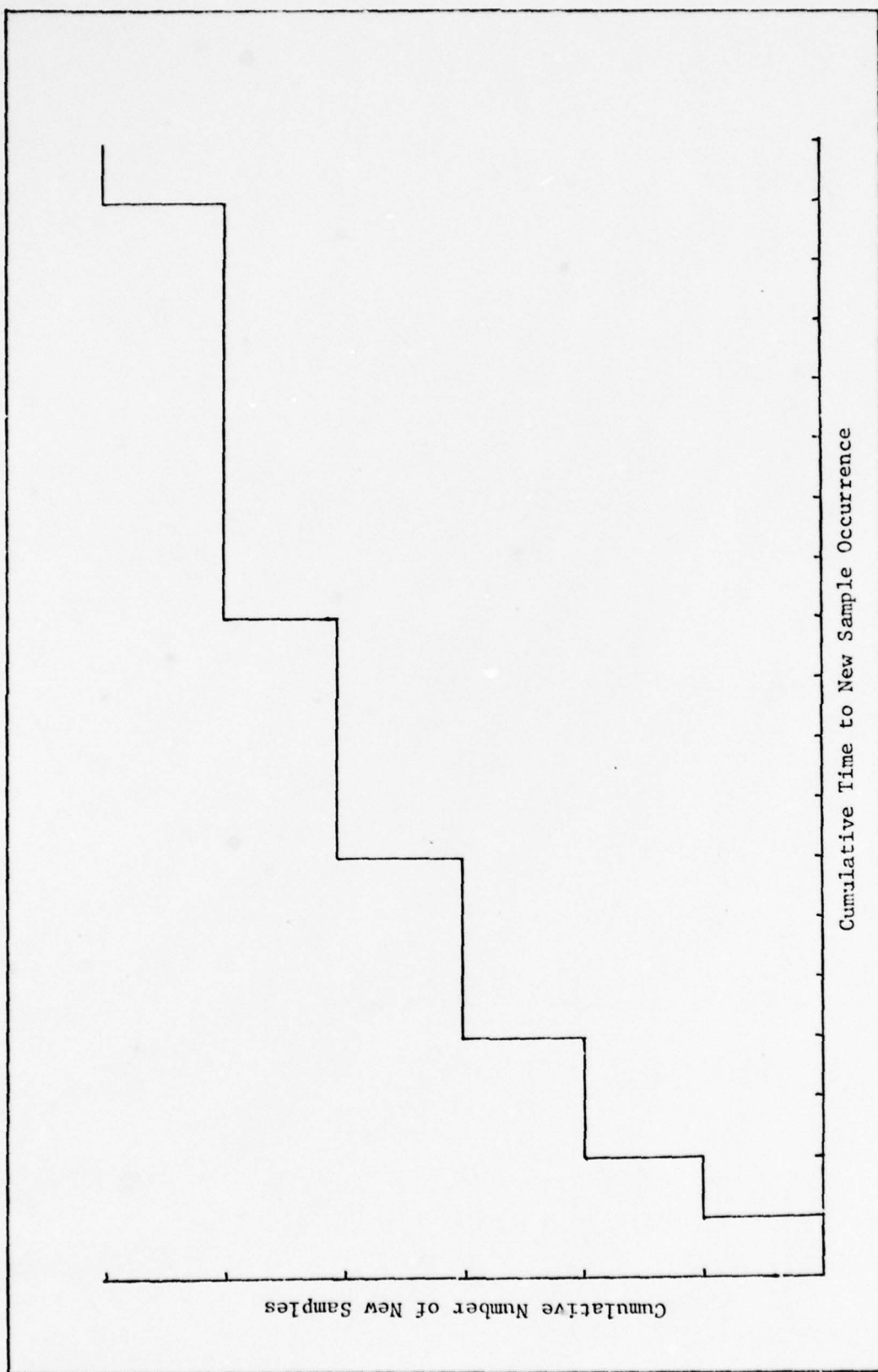


Fig. 38. Empirical Distribution

The likelihood ratio test can then be used to determine which of the remaining distributions provides the best fit. Using this distribution, an estimate \hat{N} for N can be obtained. \hat{N} can then be substituted for N in

$$z \geq N(1-\alpha) \quad (95)$$

thereby giving the tester the number of new samples he must obtain to have a confidence of $1-\alpha$ that the extremum has been encountered.

Obtaining a "reliable" estimate \hat{N} for N can require a great deal of computation, so it is not recommended that it be done after each sample is entered in the population of local maxima. Rather, it can be done after a certain pattern of "old" and "new" samples occurs. Such a pattern might be a "new", followed by at least ten "olds", followed by another "new".

If the problem is considered to be one strictly dealing with discrete-valued functions, then the problem is analogous to one of drawing balls from an urn. The objective is to find an estimate for the number of balls in the urn using random sampling with replacement. There are N balls in the urn (N unknown) and initially all the balls are white. A ball is drawn at random, with the order of the draw and the color of the ball being recorded. The ball is then painted black (regardless of its color) and replaced in the urn. This process of drawing balls one at a time, recording their color and the sample order, painting them, and then returning them to the urn is repeated until, with some level of confidence $1-\alpha$ (a different from α), it can be determined that

$$\hat{N}(1-e) \leq N \leq \hat{N}(1+e) \quad (96)$$

where \hat{N} is an estimate for N . The frequency distribution covering the pattern of random samples is unknown. Using Monte Carlo techniques, a good approximation to the distribution can be obtained empirically. To

obtain the estimated frequency distribution, a population with a known number N of white balls is constructed. Samples are drawn randomly using the painting and replacement criteria until a specified number of white balls have been drawn. Sampling is then terminated and the distribution estimated. Based on the empirical distribution, \hat{N} can be obtained as an estimate for N , which is known. The process can then be repeated using different size populations to verify that the method of determining \hat{N} still produces a good approximation to N . Once \hat{N} can be determined with a confidence of $1-\alpha$, then the calculations for obtaining \hat{N} can be applied to the sequence of "new" and "old" samples in the population of local maxima to determine an estimate of the total number of "new" samples which might occur. Then, using

$$z \geq \hat{N}(1-\alpha) \quad (97)$$

the stopping criterion of finding at least z "new" samples can be determined.

Summary. The quasilinearization method employs a suboptimal search technique in an attempt to locate the local maxima of the error function ER. Several important assumptions were made in developing the quasilinearization method:

- an upper bound exists for the function ER,
- this upper bound can be approximated by constructing linear splines,
- two points can always be found which lie arbitrarily close to the surface (upper bound),
- there are a finite number of local maxima which can be found,
- each local maximum has an equally likely chance of occurring,
- and

- one local maximum is larger than the rest and is the extremum.

Local maxima are found by starting at random locations and then alternately moving to the surface, obtaining splines, and jumping until splines can no longer be obtained. Each local maximum thus obtained is entered into a population of other local maxima and is marked as being either a "new" maximum or an "old" one. Whenever a specified pattern of "new" and "old" samples have occurred, the population of local maxima is statistically analyzed to obtain an estimate of the total number of "new" local maxima which can theoretically be found. This estimate, when multiplied by $1-\alpha$, gives the tester the number of "new" local maxima to obtain before stopping. If, at any time, a value $f(Q)$ is obtained which exceeds the specified error bounds, the process stops and the tester is able to reject the routine being analyzed.

VII Conclusions and Recommendations

The purpose of this investigation is to develop tools and techniques which can be used to determine if a given computer can solve a given avionics signal processing problem within certain specified error and time tolerances. Specifically, the following goals were defined:

- develop a tool to simulate accurately the computational characteristics of any digital processor
- produce common library routines which are optimal in the sense that they try to maximize both absolute and relative accuracy and at the same time minimize the number of instructions (especially multiplications) required
- demonstrate the effectiveness of the simulation tool mentioned above to analyze the error characteristics of a given class of algorithms and associated routines.

Each of these three goals were met with one exception. Only routines for the square root, sine, and cosine functions were developed and analyzed. Avionics algorithms also incorporate the arctangent function, which was not analyzed.

Conclusions

The n-bit simulator can be used effectively in a forward error analysis. Several enhancements were made to the n-bit simulator to simulate more accurately the numerical effects caused by finite word-lengths of various computers. Numerical routines can be evaluated by first coding them in FORTRAN as two identical subroutines controlled by a driver module. The first subroutine uses CDC library routines and, together with the driver, is compiled without being preprocessed by the

n-bit simulator. The second subroutine and the associated function subprograms are preprocessed by the n-bit simulator before being compiled. The driver module determines the error characteristics by comparing the values returned from the two subroutines.

A simulation analysis based on Monte Carlo techniques requires that sequences of events be generated where each sequence obeys a probability law governing a particular component of the random behavior in question. One law commonly encountered in simulation studies assumes that events in a sequence are independent and identically distributed (Ref 19:167). Therefore, when pseudo-random numbers are utilized, they should be tested for randomness, correlation, and goodness of fit to a specified distribution.

Since floating-point numbers are not uniformly dense on the entire number line, but only over short subintervals, trial numbers are constructed which provide a pseudo-random sampling of the representable numbers, with each number in the interval of consideration being equally-likely to be drawn. Extra trial numbers can also be constructed to test the error characteristics near the interval boundaries. Error plots which plot maximum errors in short plotting intervals against each input variable can also be utilized to aid visually in determining error characteristics.

When testing avionics algorithms, special mathematical approximations for sine, cosine, and arctangent should be used to reflect more accurately the error characteristics of the software as if it were actually executed on the computer being simulated. Minimax polynomial approximations can be used to minimize the maximum error (either absolute or relative).

Only one avionics routine (Bearing To Go) was analyzed. Since the Bearing To Go routine is a four-variate function, techniques demonstrated with this routine can easily be extended to other multivariate function routines. Error plots clearly show that the specified error bounds were exceeded. At least some of the input values which cause these error specifications to be exceeded might be considered as never occurring in an operational environment. The routine, Bearing To Go, was not evaluated using only those inputs which might be expected to occur in an operational environment.

The quasilinearization method as proposed is an heuristic approach to finding local maxima. Since the method uses searches which are conducted only in directions parallel to the variable axes, the maximum value of the error function found on any one test may not be a true local maximum. Sharp ridge lines can prevent the proposed search technique from finding true local maxima. Each test consists of first determining a random starting point and then performing a series of events called moving to the surface, obtaining a spline, and jumping to produce a monotone nondecreasing sequence of values of the error (either absolute or relative) function. The maximum value obtained on each test and its location with respect to the input variables is entered in a population to be statistically analyzed to determine the stopping criteria. Testing stops when the tester reaches a desired level of confidence that the largest local maximum discovered is the extremum, or when the error limitations are exceeded.

Recommendations

There are four different areas which are recommended for further development. The n-bit simulator needs four enhancements to make it

more flexible for use in forward error analyses and perturbation analyses. The special purpose mathematical routines used in avionics programs merit more study (especially the arctangent function approximation). The specifications (Ref 57) for the avionics algorithms should be more explicit in defining accuracy, and once the limits on the input variables have been more clearly defined, more study needs to be done on the accuracy attainable in the AN/AYK-15A processor. Extensive work is needed to develop rigorously the modified quasilinearization method, with particular emphasis on attaining and then verifying the presence of local maxima and also on the movement, or jumping, techniques.

N-bit Simulator. The first two enhancements to be considered for the n-bit simulator are those recommended by Klein (Ref 37). The first recommendation is to provide more flexibility in handling overflow, and the second recommendation is to modify the n-bit preprocessor to substitute in-line code instead of subroutine calls, thereby improving the execution time of the n-bit simulated program.

The third enhancement is to add to the existing n-bit simulator subroutines another subroutine which will generate pseudo-random numbers which represent a random sampling of all the floating-point numbers representable in a given range, and therefore are uniformly-distributed equally over each subinterval which contains numbers with the same exponent.

The fourth recommendation is to add to the n-bit simulator subroutines a subroutine which returns floating-point numbers adjacent to one which is input. Inputs to this subroutine might be a starting number, direction to go (positive or negative), and number of values to return. Outputs from this subroutine would include the values requested and a

status flag indicating either no error or overflow (either positive or negative). The presence of zero as a value returned can also be indicated.

Mathematical Routines. All the special-purpose mathematical routines investigated used extended-precision to perform argument reductions and single-precision to perform the polynomial approximations (in the case of the trigonometric approximations) or Newton iterations (in the case of the square root approximations). The trigonometric approximations, and especially the arctangent function, merit further study using extended precision and/or higher-order polynomials for use in an avionics environment.

Avionics Routines. Error tolerance limits need to be defined more clearly in the specifications. Errors were found which exceeded the limits in the specifications, and although the inputs used clearly fall within the variable ranges defined, it is highly unlikely that these values would occur in an operational environment. Without specifications which also show the limitations placed on the relationships between variables, correct analyses cannot be performed. One example of this would be specifying certain combinations of inputs which do not need to meet the normal error specifications. For the Bearing To Go routine, this might mean that waypoints within a 1-mile radius or beyond a 3000-mile radius of the aircraft would not be subject to the error bounds specified for those waypoints which lie between 1 and 3000 miles from the aircraft. The numbers 1 and 3000 are used here strictly for the purposes of the example.

Quasilinearization Method. The quasilinearization as proposed should be tested on both single and multivariate functions to determine

its merit relative to other optimization and suboptimization techniques. Various acceleration techniques should be investigated to determine their effect on the convergence of the proposed method. The concept of moving to the surface should be examined more closely to determine the number of points to examine in each direction. As the probability of being within an arbitrarily small distance from the "surface" increases, better jumping techniques should be able to be utilized. Searching techniques should be incorporated which are not limited to directions parallel to an axis. This would allow ridge-following algorithms to be developed, thereby increasing the probability that the extreme value returned by any single test is in fact a true local maximum. As the probability of obtaining true local maxima goes up, the number of "new" samples entered in the sample population should decrease correspondingly, thereby allowing for fewer tests to be conducted.

Better methods for analyzing the sample population should be investigated. As searching techniques become more complex, the population should be analyzed more frequently, hence the need for a more efficient algorithm to determine the stopping criteria. Methods proposed should be able to be applied to the ball and urn problem discussed in Chapter 6.

Bibliography

1. Abd-alla, Abd-elfattah M., and Arnold C. Meltzer. Principles of Digital Computer Design, Vol. I. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1976.
2. Air Force Avionics Lab. DAIS Prime Item Development Specification for AN/AYK-15A Digital Processor, Part I. Wright-Patterson AFB, Ohio: Air Force Avionics Lab, 1 September 1978.
3. Ashenhurst, R. L. "Number Representation and Significance Monitoring," Mathematical Software, edited by J. R. Rice, New York: Academic Press, 1971, pp 67-92.
4. Beveridge, Gordon S. G. and Robert S. Schechter. Optimization: Theory and Practice. New York: McGraw-Hill Book Company, 1970.
5. Brent, R. P. "A FORTRAN Multiple-Precision Arithmetic Package," Report, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, May 1976.
6. Bright, H. S., and I. J. Cole. "A Method of Testing Programs for Data Sensitivity," Program Test Methods, edited by William C. Hetzel. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1973, pp 143-162.
7. Britting, Kenneth R. Inertial Navigation Systems Analysis. New York: John Wiley & Sons, Inc., 1971.
8. Buck, R. C. Advanced Calculus. New York: McGraw-Hill Book Company, 1965.
9. Chartres, Bruce A. "Automatic Controlled Precision Calculations," Journal of the Association for Computing Machinery, 13, 386-403 (July, 1966).
10. Cody, W. J. "Performance Testing of Function Subroutines," Proceedings, Spring Joint Computer Conference, 34, 759-763 (1969)
11. Cody, W. J. "The Evaluation of Mathematical Software," Program Test Methods, edited by William C. Hetzel. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1973, pp 121-133.
12. Cody, W. J. "Static and Dynamic Numerical Characteristics of Floating-Point Arithmetic," IEEE Transactions on Computers, C-22, 598-601 (June, 1973).
13. Cody, W. J. "The Challenge in Numerical Software for Microcomputers," IEEE Proceedings, First Annual Rocky Mountain Symposium on Microcomputers: Systems, Software, Architecture, 1-23 (September 1977).
14. Control Data Corporation. FORTRAN Common Library Mathematical Routines (60387900). Sunnyvale, California: Control Data Corporation, 1976.

15. Control Data Corporation. 6600 Central Processor, Vol. II, Functional Units (60239700). Sunnyvale, California: Control Data Corporation, 1976.
16. Dahlquist, Germund, and Ake Bjorck. Numerical Methods. Englewood Cliffs, New Jersey: Prentice-Hall, 1974.
17. Dunham, C. B. "Nonstandard Arithmetic", Mathematical Software, edited by J. R. Rice, New York: Academic Press, 1971, pp 105-111.
18. Fike, C. T. Computer Evaluation of Mathematical Functions. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1968.
19. Fishman, George S. Concepts and Methods in Discrete Event Digital Simulation. New York: John Wiley & Sons, Inc., 1973.
20. Forsythe, George E. Pitfalls in Computation, or Why a Math Book Isn't Enough. Technical Report No. CS 147. Stanford, California: Stanford University, January, 1970. (AD 699 897).
21. Forsythe, George E., Michael A. Malcolm, and Cleve B. Moler. Computer Methods for Mathematical Computations. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1977.
22. Garner, Harvey L. "A Survey of Some Recent Contributions to Computer Arithmetic", IEEE Transactions on Computers, C-25, 1277-1282 (December, 1976).
23. General Dynamics Corporation. Computer Program Development Specification for the F-16 Fire Control Computer Operational Flight Program, CDRL Item AOOE, (16ZE011-1C). Fort Worth, Texas: Fort Worth Division, General Dynamics Corporation, 1977.
24. Ghosh, B. K. Sequential Tests of Statistical Hypothesis. Reading, Massachusetts: Addison-Wesley Publishing Company, Inc., 1970.
25. Ginsberg, Myron, and Dennis J. Frailey. "The Design and Use of a Floating Point (Software) Simulator for Testing the Arithmetic Behavior of Mathematical Software", IEEE Third Symposium on Computer Arithmetic, 56-63 (November, 1975).
26. Ginsberg, Myron. "Numerical Influences on the Design of Floating-Point Arithmetic for Microcomputers", IEEE Proceedings, First Annual Rocky Mountain Symposium on Microcomputers: Systems, Software, Architecture, 25-72 (September, 1977).
27. Goldberg, I. Bennett. "27 Bits Are Not Enough for 8-Digit Accuracy", Communications of the Association for Computing Machinery, 10, 105-106 (February, 1967).
28. Gumbel, E. J. Statistics of Extremes. New York: Columbia University Press, 1958.

29. Hammer, C. "Statistical Validation of Mathematical Computer Routines", Proceedings, Spring Joint Computer Conference, 30, 331-333 (1967).
30. Hamming, R. W. Digital Filters. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1977.
31. Hansen, E. (Ed.). Topics in Interval Analysis. London: Oxford University Press, 1969.
32. Hart, J. C., et al. Computer Approximations, New York: John Wiley & Sons, Inc., 1968.
33. Hemker, P. W., et al. Single and Double-Length Computation of Elementary Functions. Amsterdam: Mathematical Centre, October, 1973 (AD 914 647).
34. Hollander, Myles, and Douglas A. Wolfe. Nonparametric Statistical Methods. New York: John Wiley & Sons, Inc., 1973.
35. Hull, T. E., and J. J. Hofbauer. "Language Facilities for Multiple Precision Floating Point Computation with Examples and the Description of a Preprocessor", Technical Report No. 63, Department of Computer Science, University of Toronto, Ontario, Canada, February, 1974.
36. Kaneko, Toyohisa, and Bede Liu. "On Local Roundoff Errors in Floating Point Arithmetic", Journal of the Association for Computing Machinery, 20, 391-398 (July, 1973).
37. Klein, Gary A. Software Tool(s) for Evaluating the Effects of Finite Wordlength. MS Thesis. Wright-Patterson AFB, Ohio: Air Force Institute of Technology, December 1977. (AD A055 777).
38. Knuth, D. E. The Art of Computer Programming: Seminumerical Algorithms, Vol. 2. Reading, Massachusetts: Addison-Wesley Publishing Company, Inc., 1969.
39. Kuck, David J., Douglass S. Parker, and Ahmed H. Sameh. "Analysis of Rounding Methods in Floating Point Arithmetic", IEEE Transactions on Computers, C-26, 643-650 (July 1977).
40. Kuck, D. J., D. S. Parker, and A. H. Sameh. "ROM-Rounding: A New Rounding Scheme", IEEE Third Symposium on Computer Arithmetic, 67-72 (November, 1975).
41. Kuki, H. "Comments on the ANL Evaluation of OS/360 FORTRAN Math Function Library", SHARE Secretary Distribution 169, C-4773, 47-53 (1967).
42. Kuki, H., and J. Ascoly. "FORTRAN Extended-Precision Library", IBM Systems Journal, 10, 39-61 (1971).

43. Kuki, H., and W. J. Cody. "A Statistical Study of the Accuracy of Floating Point Number Systems", Communications of the Association for Computing Machinery, 16, 223-230 (April 1973).
44. LaFara, Robert L. Computer Methods for Science and Engineering. Rochelle Park, New Jersey: Hayden Book Company, Inc., 1973.
45. Lewis, T. G. Distribution Sampling for Computer Simulation. Lexington, Massachusetts: D. C. Heaton and Company, 1975.
46. Maehly, Hans J. "Methods for Fitting Rational Approximations, Part I: Telescoping Procedures for Continued Fractions", Journal of the Association of Computing Machinery, 7, 150-162 (April 1960).
47. Miller, Irwin, and John E. Freund. Probability and Statistics for Engineers. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1977.
48. Miller, Webb. "Computer Search for Numerical Instability", Journal of the Association for Computing Machinery, 22: 512-521 (October 1975).
49. MIL-STD-1750. Airborne Computer Instruction Set. Wright-Patterson AFB, Ohio: Aeronautical Systems Division/ENAI, 1 June 1978.
50. Moore, Ramon E. Interval Analysis. Englewood Cliffs, New Jersey: Prentice-Hall, 1966.
51. Moore, Ramon E. Mathematical Elements of Scientific Computing. New York, New York: Holt, Rinehart, and Winston. Inc., 1975.
52. Newbery, A. C. R., and A. P. Leigh. "Consistency Tests for Elementary Functions", Proceedings, Fall Joint Computer Conference, 39, 419-422 (1971).
53. Nickel, K. (Ed.). Interval Mathematics, Vol. 29, Lecture Notes in Computer Science. Berlin: Springer-Verlog, 1975.
54. Noether, G. E. Elements of Nonparametric Statistics. New York: John Wiley & Sons, Inc., 1967.
55. Oppenheim, Allan V. and Ronald W. Schaffer. Digital Signal Processing. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1975.
56. Ruckdeschel, Fred. "Functional Approximations", BYTE, (November, 1978).
57. Singer Company, The. Computer Program Development Specification for F-16 Inertial Navigation Set Operational Flight Program (Y240A085A0200). Kearfott Division, The Singer Company, 1976.
58. Stone, Harold S., et al. Introduction to Computer Architecture. Chicago, Illinois: Science Research Associates, Inc., 1975.

59. Stoutemyer, D. R. "Automatic Error Analysis Using Computer Algebraic Manipulation", ACM Transactions on Mathematical Software, 3, 26-43 (March, 1977).
60. Tienari, M. Varying Length Floating Point Arithmetic: A Necessary Tool for the Numerical Analyst. Technical Report No. 62. Stanford, California: Computer Science Department, Stanford University, 1967.
61. Tsao, Nai-Kuan. "On the Distribution of Significant Digits and Roundoff Errors", Communications of the Association for Computing Machinery, 17, 269-271 (May 1974).
62. Tsao, Nai-Kuan. "Some a Posteriori Error Bounds in Floating-Point Computations", Journal of the Association for Computing Machinery, 21, 6-17 (January 1974).
63. Vignes, J., and M. LaPorte. "Error Analysis in Computing", IFIP Congress 74, 3: 610-614 (1974).
64. Wilde, Douglass J. Optimum Seeking Methods. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1964.
65. Wilkinson, J. H. "Modern Error Analysis", SIAM Review, 13, 548-568 (October, 1971).
66. Wilkinson, J. H. Rounding Errors in Algebraic Processes. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1963.
67. Wingo, D. R. "Maximum Likelihood Estimation of the Parameters of the Weibull Distribution by Modified Quasilinearization", IEEE Transactions on Reliability, R-21, 89-93 (1972).
68. Wyatt, W. T., Jr., D. W. Lozier, and D. J. Orser. "A Portable Extended Precision Arithmetic Package and Library with FORTRAN Precompiler", ACM Transactions on Mathematical Software, 2, 209-231 (September 1976).

Appendix A

User's Manual for the First Revision of the N-bit Simulator

By using the first revision of the n-bit simulation tool, a user can evaluate the numerical effects that various computer architectures can have upon a FORTRAN-programmed algorithm. The simulator was designed to be executed on either CDC 6600 or CYBER 74 computer systems and requires that programs being executed by the simulator be expressed in the FORTRAN IV (extended) programming language with a few restrictions.

This user's manual will describe the ten user options available for describing different computer architectures and the control cards for executing the first revision of the n-bit simulator. The programming restrictions and considerations applicable to the original simulator still apply (Ref 37).

User Options

To provide the user with greater flexibility in simulating the effects of various computer architectures, six of the seven options of the original n-bit simulator and four additional options, for a total of ten user options, were incorporated into the first revision of the simulator. These changes require only minor modifications on the part of the user.

First, two arrays must now be dimensioned in each routine. The array named KEY must be dimensioned to eight (e.g. DIMENSION KEY(8)) and the array named TKEY must be dimensioned to four (e.g. DIMENSION TKEY(4)). The array TKEY holds the floating point values to be used in the overflow and underflow checks, and the array KEY holds the fixed point overflow values and the user options.

Second, the SETNBIT routine which is called as the first executable statement and wherever else desired now has twelve parameters, two of which are user options, and the two arrays KEY and TKEY. The required form of the SETNBIT subroutine call is shown below, with each # representing a user option.

```
CALL SETNBIT (#1,#2,#3,#4,#5,#6,#7,#8,#9,#10,KEY,TKEY)
```

Option #1. The first option allows the user to specify the number of bits per wordlength (from 4 to 60) used for floating-point arithmetic.

Option #2. The second option allows the user to specify the number of bits in the mantissa, not counting the sign bit, for floating-point words (from 1 to 48).

Option #3. This option allows the user to specify the number of guard digits to be used during intermediate floating-point calculations. This option replaced the single, double, or triple precision option of the original simulator and has no effect on fixed-point calculations. All floating-point variable assignments are still made using the word-length specified in the first option.

Option #4. Option four allows the user to specify the number of bits to be used in the floating-point exponent, including the exponent sign bit (from 2 to 11). The sum the values for option four and option two must be one less than that for option one.

Option #5. The fifth option allows the user to specify the position of the implied binary point with respect to the mantissa. The value 0 positions the point to the left of the mantissa (making the mantissa a fraction) and the value 1 positions the point to the right of the mantissa (making the mantissa an integer).

Option #6. Option six allows the user to specify whether rounding

or truncation will be performed after each operation. The value 0 indicates truncation and the value 1 indicates rounding.

Option #7. This option was added to allow the user to specify whether the computer being simulated uses sign plus one's complement or sign plus two's complement arithmetic. Sign-magnitude machines should be simulated by specifying sign plus one's complement. The value 1 is used to specify sign plus one's complement and the value 2 is used to specify sign plus two's complement arithmetic.

Option #8. Option eight was added to allow the user to specify whether the machine being simulated is a binary, quaternary, octal, or hexadecimal machine. The value 1 is used to select binary, the value 2 for quaternary, the value 3 for octal, and the value 4 for hexadecimal.

Option #9. The ninth option allows the user to suppress the printing of overflow and underflow messages. These messages specify whether overflow or underflow occurred, whether positive or negative, and the routine and line number where it occurred. If overflow occurs on the actual CDC word, the program will terminate.

Option #10. This option was added to allow the user to specify the wordlength used for fixed-point calculations. This will allow for valid overflow checking for both fixed-point and floating-point without having to call SETNBIT whenever switching from one type to the other. The maximum value allowed is 48.

An example of a typical SETNBIT subroutine call is

```
CALL SETNBIT (32,23,0,8,0,0,2,1,1,16,KEY,TKEY)
```

The argument values shown specify a 32-bit floating-point word, broken down into 1 sign bit, 8 exponent bits, and 23 mantissa bits, with no guard bits. The binary point is on the left of the mantissa, and the

binary machine does two's complement arithmetic with truncation. Overflow and underflow messages are to be printed, and fixed-point calculations are performed using a 16-bit wordlength. If double-precision floating-point variables are used, the first and second options should be changed to reflect the correct wordlength, thus allowing for double-precision calculation and storage. This must be distinguished from double-precision calculation with single-precision storage, which can be accomplished by specifying the correct number of guard digits.

Control Cards

A very basic understanding of how the simulator works helps in constructing job control cards to simulate a program. The preprocessor, which is executed first, reads as data input the FORTRAN program to be simulated, modifies the assignment statements, and writes out a file containing the source of the modified program. The program on this file must then be compiled and executed. As it executes, it will call the subroutines which are loaded with the objects of the compiled program.

There are many different ways to build a deck to execute a program using the n-bit simulator. One very simple deck setup using source cards for the preprocessor, the subroutines, and the FORTRAN program to be simulated, is shown below.

```
FTN,L=DUMMY1.  
LGO,,,NBIT.  
REWIND,NBIT.  
FTN,L=DUMMY2,B=SUBS.  
REWIND,SUBS.  
FTN,I=NBIT,B=PROG.  
REWIND,PROG.  
LOAD,PROG,SUBS.  
EXECUTE.
```

```

7/8/9 Card
***Preprocessor source cards***
7/8/9 Card
***FORTRAN program source cards***
7/8/9 Card
***Subroutines source cards***
7/8/9 Card
6/7/8/9 Card

```

Another simple way to execute a program using the simulator involves using object decks for the preprocessor and the subroutines. A deck setup to execute this way is shown below.

```

INPUT,,,NBIT.
REWIND,NBIT.
FTN,I=NBIT.
LOAD,LGO,INPUT.
EXECUTE.
7/8/9 Card
***Preprocessor objects***
7/8/9 Card
***FORTRAN program source cards***
7/8/9 Card
***Subroutine objects***
7/8/9 Card
6/7/8/9 Card

```

The above shown job setups involve using large card decks which are rather cumbersome. If several executions are planned, the objects for the preprocessor and subroutines should be put on permanent files where they could be accessed using either batch or remote job entry. The deck setup shown below will store the preprocessor and subroutine objects on permanent files.

```

REQUEST,OBJECT1,*PF.
REQUEST,OBJECT2,*PF.

```

```

COPYBF, INPUT, OBJECT1.
COPYBF, INPUT, OBJECT2.
REWIND, OBJECT1.
REWIND, OBJECT2.
CATALOG, OBJECT1, NBITSIM, CY=1, RP=120.
CATALOG, OBJECT2, NBITSIM, CY=2, RP=120.
7/8/9 Card
***Preprocessor objects***
7/8/9 Card
***Subroutine objects***
7/8/9 Card
6/7/8/9 Card

```

Once the preprocessor and subroutine objects are on permanent files, they can be accessed by using the following deck setup.

```

ATTACH, OBJECT1, NBITSIM, CY=1.
ATTACH, OBJECT2, NBITSIM, CY=2.
REWIND, OBJECT1.
REWIND, OBJECT2.
OBJECT1,,, NBIT.
REWIND, NBIT.
FIN, I=NBIT.
LOAD, LGO, OBJECT2.
EXECUTE.
7/8/9 Card
***FORTRAN program source cards***
7/8/9 Card
6/7/8/9 Card

```

The user options added to the first revision make it even more costly to simulate a program. Core requirements changed only to the extent that the subroutines are slightly larger and each call to the subroutines contains one additional parameter. It is conjectured that execution time will increase approximately twenty percent, since the subroutines are longer and more complex.

Appendix B

N-bit Simulator Subroutines Source Listing

The concept of using 22 n-bit subroutines to handle each combination of data types was retained from the original simulator, although the code in the subroutines was extensively modified. The SETNBIT subroutine was also extensively modified to give the user more flexibility with the simulator. Three other subroutines called ROUNDER, ONETRNC, and TWOTRNC were added to handle all rounding, sign plus one's complement truncation, and sign plus two's complement truncation. In the program listing below, the subroutines are arranged into four major categories. The SETNBIT subroutine comes first, followed by the special rounding and truncation routines. The third group is composed of those subroutines which handle only integer variables, and the fourth group contains those subroutines which handle at least one real variable.

```

SUBROUTINE SETNB11(NBITS, MANTISA, IGUARD, IEXPNT, IPTPOS, IRNDTR,
* ICMWNO, ITYPE, MESSGS, IFIXD, KEY, TKEY)
DIMENSION KEY(8)
DIMENSION TKEY(4)

```

```

DATA IMXPOS/37757777777777777777B/
DATA IMXNEG/40010000000000000000B/
DATA IMWPOS/00010000000000000000B/
DATA IMWNEG/77763777777777777777B/

```

EDITS SECTION

THIS SECTION EDITS THE INPUT FOR LEGAL VALUES

```

NBITS    NUMBER OF BITS IN FLOATING POINT WORD
MANTISA   MANTISSA LENGTH W/O SIGN OF GUARD DIGITS
IGUARD    NUMBER OF GUARD DIGITS USED FOR INTERMEDIATE VALUES
IEXPNT    EXPONENT LENGTH W/ SIGN BIT
IPTPOS    0 = LEFT      1 = RIGHT
IRNDTR    0 = TRUNCATE  1 = ROUND
ICMWNO    1 = ONES COMPLEMENT  2 = TWOS COMPLEMENT
ITYPE     1 = BINARY  2 = QUATERNARY  3 = OCTAL  4 = HEXADECIMAL
MESSGS    0 = SUPPRESSION  1 = PRINT
IFIXD     NUMBER OF BITS IN FIXED POINT WORD

```

```

KEY       ARRAY TO FILL FOR OTHER SUBROUTINES
KEY(1)    MAX POSITIVE FIXED POINT VALUE
KEY(2)    MAX NEGATIVE FIXED POINT VALUE
KEY(3)    IRNDTR
KEY(4)    ICMWNO
KEY(5)    MESSGS
KEY(6)    FINAL MANTISSA BITS SAVED (MANTISA)
KEY(7)    INTERMEDIATE EXTRA BITS SAVED (IGUARD)
KEY(8)    ITYPE

```

```

TKEY      ARRAY TO FILL FOR OTHER SUBROUTINES
TKEY(1)    MAX POSITIVE FLOATING POINT VALUE
TKEY(2)    MAX NEGATIVE FLOATING POINT VALUE
TKEY(3)    MIN POSITIVE FLOATING POINT VALUE
TKEY(4)    MIN NEGATIVE FLOATING POINT VALUE

```

```

IF (NBITS.GT.60) GO TO 210
IF ((MANTISA.GT.45).OR.(MANTISA.LT.1)) GO TO 215
IF ((IRNDTR.NE.0).AND.(IRNDTR.NE.1)) GO TO 220
IF ((NBITS.EQ.67).AND.(IRNDTR.EQ.1)) GO TO 225
IF ((ICMWNO.NE.1).AND.(ICMWNO.NE.2)) GO TO 230
IF ((IGUARD.LT.0).OR.(IGUARD.GT.(48-MANTISA))) GO TO 235
IF ((ITYPE.NE.1).AND.(ITYPE.NE.2).AND.(ITYPE.NE.3).AND.
* (ITYPE.NE.4)) GO TO 240
IF ((MESSGS.NE.0).AND.(MESSGS.NE.1)) GO TO 245
IF ((IEXPNT.GT.1).OR.(IEXPNT.LT.1)) GO TO 250
IF ((IEXPNT + MANTISA + 1).NE.NBITS) GO TO 255
IF ((IPTPOS.NE.0).AND.(IPTPOS.NE.1)) GO TO 260
IF ((IFIXD.GT.46) GO TO 265

```

END EDITS SECTION

```

KEY(1) = SHIFT(MASK(1), IFIXD) - 1

```

```

KEY(2) = -KEY(1)
IF ((ICNTWO.EQ.2).AND.(IFIXD.NE.48)) KEY(2) = KEY(2) - 1
KEY(3) = IRNDTR
KEY(4) = IONTWO
KEY(5) = MESSGS
KEY(6) = MANTSA
KEY(7)=IGUARD
KEY(8) = ITYPE

C
C      FILL FLOATING POINT OVERFLOW AND UNDERFLOW CHECKS
C      IF UNDERFLOW RESULTS, ZERO WILL BE USED
C      IF OVERFLOW RESULTS, MAX CDC VALUES WILL BE USED
C
C      THESE CHECKS ARE GOOD ONLY FOR COMPUTERS WITH UNBIASED EXPONENTS
C
C      START WITH MAX AND MIN CDC VALUES
TKEY(1)=IMXPOS
TKEY(2)=IMXNEG
TKEY(3)=IMNPOS
TKEY(4)=IMNNEG
C      CREATE MASK OF MANTISSA BITS TO SAVE
TEMP1=SHIFT(MASK(MANTSA),48)
C      CREATE MAX EXPONENT
ITEMP=-48+ITYPE*(2**(IEXPNT-1))
C      ADJUST FOR LOCATION OF BINARY POINT
IF (IPTPOS.EQ.1) ITEMP=ITEMP+MANTSA
JTEMP=1024
C      TRANSFER IMPLIES CDC VALUES ARE USED
IF (ITEMP.GE.JTEMP) GO TO 50
C      CODE TO HANDLE CDC EXPONENT HANDLING
ITEMP IS BINARY EXPONENT AS INTEGER
IF (ITEMP.LT.0) JTEMP=JTEMP-1
ITEMP=ITEMP+JTEMP
C      SHIFT EXPONENT INTO POSITION
TEMP2=SHIFT(ITEMP,43)
C      CREATE MAX POSITIVE FLOATING POINT VALUE
TKEY(1)=TEMP2.OR.TEMP1
C      CREATE MAX NEGATIVE FLOATING POINT VALUE
TKEY(2)=-TKEY(1)
C      SKIP SPECIAL CASE FOR TWOS COMPLEMENT
IF (ICNTWO.EQ.1) GO TO 50
C      CREATE UNNORMALIZED ADJUSTMENT
TEMP3=SHIFT(MASK(1),49-MANTSA)
TEMP4=TEMP2.OR.TEMP3
C      ADJUST TKEY(2)
TKEY(2)=TKEY(2)-TEMP4
C      CREATE MANTISSA FOR UNDERFLOW
50 TEMP1=SHIFT(MASK(1),48)
C      CREATE MIN EXPONENT (MAX NEGATIVE)
ITEMP=-48-ITYPE*(2**(IEXPNT-1))
C      ADJUST FOR TWOS COMPLEMENT
IF (ICNTWO.EQ.1) ITEMP=ITEMP+ITYPE
C      ADJUST FOR LOCATION OF BINARY POINT
IF (IPTPOS.EQ.1) ITEMP=ITEMP+MANTSA
JTEMP=1023
C      RETURN IMPLIES CDC VALUES ARE USED
IF (ITEMP.LT.-JTEMP) RETURN

```

```

C      CODE TO HANDLE CISC EXPONENT HANDLING
C      ITEMF IS BINARY EXPONENT AS INTEGER
ITEMF=ITEMF*JTEMP
C      SHIFT EXPONENT INTO POSITION
TEMP2=SHIFT(ITEMF,49)
C      CREATE MIN POSITIVE FLOATING POINT VALUE
TKEY(3)=TEMP2.OR.TEMP1
C      CREATE MIN NEGATIVE FLOATING POINT VALUE
TKEY(4)=-TKEY(3)
C      SKIP SPECIAL CASE FOR TWOS COMPLEMENT
IF (IONTWO.EQ.1) RETURN
C      CREATE UNNORMALIZED ADJUSTMENT
TEMP2=SHIFT(MASK(1),49-MANTSA)
TEMP4=TEMP2.OR.TEMP3
C      ADJUST TKEY(4)
TKEY(4)=TKEY(4)-TEMP4
RETURN
C
C
210 PRINT*,"NBITS TOO BIG", NBITS
STOP
215 PRINT*,"BAD MANTISSA LENGTH ",MANTSA
STOP
220 PRINT*,"BAD ROUND/TRUNCATE OPTION ",IRNDTR
STOP
225 PRINT*,"CANT ROUND W/ 60 BITS ",NBITS
STOP
230 PRINT*,"MACHINE ONLY DOES ONES OR TWOS COMP STUFF"
PRINT*,"FOR SIGN-MAGNITUDE, USE ONES COMP ",IONTWO
STOP
235 PRINT*,"BAD GUARD DIGITS ",IGUARD
STOP
240 PRINT*,"BAD MACHINE TYPE SPECIFIED ",ITYPE
PRINT*,"1=BINARY 2=QUATERNARY 3=OCTAL 4=HEXADECIMAL"
STOP
245 PRINT*,"BAD MESSAGE OPTION REQUESTED ",MESSGS
STOP
250 PRINT*,"BAD EXPONENT ",IEXPNT
STOP
255 PRINT*,"EXPONENT AND MANTISA ARE NOT COMPATIBLE"
PRINT*,"MANTISA + IEXPNT + 1 = NBITS "
PRINT*,"IEXPNT = ",IEXPNT
PRINT*,"MANTISA = ",MANTISA
STOP
260 PRINT*,"BAD POINT POSITION REQUESTED ",IPIPOS
STOP
265 PRINT*,"BAD FIXED POINT WORDLENGTH ",IFIXO
STOP
END

```

```

C      SUBROUTINE ONETRNC(RRR,IFNL,KEY)
C
C      SUBROUTINE TO ADJUST ALL MANTISSAS TO PROPER LENGTH
C
C      USED WHEN SIMULATING MACHINES WHICH USE SIGN PLUS ONES COMPLEMENT
C      REPRESENTATION OF FLOATING POINT NUMBERS WITH TRUNCATION
C
C      DIMENSION KEY(8)
C      MASK TO PICK OFF CDC EXPONENT AND SIGN
C      DATA TRNC1/777700000000000000000000/
C      CHANGE EXPONENT TO INTEGER      DISCARD MANTISSA
C      IADJUST=SHIFT(RRR,-48)
C      ADJUST INTEGER FOR SIGN OF MANTISSA
C      IF (RRR.LT.0.) IADJUST=.NOT.IADJUST
C      ADJUST INTEGER FOR CDC EXPONENT REPRESENTATION SCHEME
C      IF (IADJUST.GE.1024) IADJUST=2056-IADJUST
C      IF (IADJUST.LT.1024) IADJUST=2056-IADJUST
C      ADJUSTMENT FOR MACHINE RADIX
C      JADJUST=MOD(IADJUST,KEY(8))
C      COMPUTE NUMBER OF BITS TO TRUNCATE
C      KADJUST=JADJUST+48-KEY(6)
C      ADJUST FOR KEEPING GUARD BITS ON INTERMEDIATE EXPRESSIONS
C      IF (IFNL.EQ.0) KADJUST=KADJUST-KEY(7)
C      TRUNCATE BY SHIFTING      FIRST RIGHT, THEN LEFT
C      RRR=SHIFT(SHIFT(RRR,-KADJUST),KADJUST)
C      RETURN
C      END

```



```

INTEGER FUNCTION I1ADD(I1,I2,ROUTINE,LNCNT,IFNL,KEY,TKEY)
DIMENSION TKEY(4)
DIMENSION KEY(8)
I1ADD=I1+I2
IF (LEGVAR(I1ADD).NE.0) GO TO 100
IF (I1ADD.GT.KEY(1)) GO TO 40
IF (I1ADD.LT.KEY(2)) GO TO 60
RETURN
40 I1ADD=KEY(1)
IF (KEY(5).EQ.0) RETURN
PRINT (1,240) ROUTINE,LNCNT
RETURN
60 I1ADD=KEY(2)
IF (KEY(3).EQ.0) RETURN
PRINT (1,260) ROUTINE,LNCNT
RETURN
100 PRINT*,"SORRY,TURKEY  I CANT CONTINUE"
PRINT (1,150) I1ADD
150 FORMAT (1H0,020)
240 FORMAT (1H0,"I1ADD POSITIVE OVERFLOW  ",A5,"  LINE = ",I5)
260 FORMAT (1H0,"I1ASN NEGATIVE OVERFLOW  ",A5,"  LINE = ",I5)
STOP
END
INTEGER FUNCTION I1MNS(I1,I2,ROUTINE,LNCNT,IFNL,KEY,TKEY)
DIMENSION TKEY(4)
DIMENSION KEY(8)
I1MNS=I1-I2
IF (LEGVAR(I1MNS).NE.0) GO TO 100
IF (I1MNS.GT.KEY(1)) GO TO 40
IF (I1MNS.LT.KEY(2)) GO TO 60
RETURN
40 I1MNS=KEY(1)
IF (KEY(5).EQ.0) RETURN
PRINT (1,240) ROUTINE,LNCNT
RETURN
60 I1MNS=KEY(2)
IF (KEY(3).EQ.0) RETURN
PRINT (1,260) ROUTINE,LNCNT
RETURN
100 PRINT*,"SORRY,TURKEY  I CANT CONTINUE"
PRINT (1,150) I1MNS
150 FORMAT (1H0,020)
240 FORMAT (1H0,"I1MNS POSITIVE OVERFLOW  ",A5,"  LINE = ",I5)
260 FORMAT (1H0,"I1EXP NEGATIVE OVERFLOW  ",A5,"  LINE = ",I5)
STOP
END
INTEGER FUNCTION I1MPY(I1,I2,ROUTINE,LNCNT,IFNL,KEY,TKEY)
DIMENSION TKEY(4)
DIMENSION KEY(8)
I1MPY=I1*I2
IF (LEGVAR(I1MPY).NE.0) GO TO 100
IF (I1MPY.GT.KEY(1)) GO TO 40
IF (I1MPY.LT.KEY(2)) GO TO 60
RETURN
40 I1MPY=KEY(1)
IF (KEY(5).EQ.0) RETURN
PRINT (1,240) ROUTINE,LNCNT
RETURN
60 I1MPY=KEY(2)
IF (KEY(3).EQ.0) RETURN
PRINT (1,260) ROUTINE,LNCNT
RETURN
100 PRINT*,"SORRY,TURKEY  I CANT CONTINUE"
PRINT (1,150) I1MPY
150 FORMAT (1H0,020)
240 FORMAT (1H0,"I1MPY POSITIVE OVERFLOW  ",A5,"  LINE = ",I5)
260 FORMAT (1H0,"I1DND NEGATIVE OVERFLOW  ",A5,"  LINE = ",I5)
STOP
END

```

```

INTEGER FUNCTION IIOVD(I1,I2,ROUTINE,LNCNT,IFNL,KEY,IKEY)
DIMENSION IKEY(4)
DIMENSION KEY(8)
IIOVD=I1/I2
IF (LEGVAR(IIOVD).NE.0) GO TO 100
IF (IIOVD.GT.KEY(1)) GO TO 40
IF (IIOVD.LT.KEY(2)) GO TO 60
RETURN
40 IIOVD=KEY(1)
IF (KEY(3).EQ.0) RETURN
PRINT (1,240) ROUTINE,LNCNT
RETURN
60 IIOVD=KEY(2)
IF (KEY(5).EQ.0) RETURN
PRINT (1,260) ROUTINE,LNCNT
RETURN
100 PRINT*,"SORRY,TURKEY  I CANT CONTINUE"
PRINT(1,150) IIOVD
150 FORMAT (1H0,020)
240 FORMAT (1H0,"IIOVD POSITIVE OVERFLOW  ",A3,"  LINE = ",I5)
260 FORMAT (1H0,"IIOVD NEGATIVE OVERFLOW  ",A3,"  LINE = ",I5)
STOP
END

INTEGER FUNCTION IIEXP(I1,I2,ROUTINE,LNCNT,IFNL,KEY,IKEY)
DIMENSION IKEY(4)
DIMENSION KEY(8)
IIEXP=I1*I2
IF (LEGVAR(IIEXP).NE.0) GO TO 100
IF (IIEXP.GT.KEY(1)) GO TO 40
IF (IIEXP.LT.KEY(2)) GO TO 60
RETURN
40 IIEXP=KEY(1)
IF (KEY(3).EQ.0) RETURN
PRINT (1,240) ROUTINE,LNCNT
RETURN
60 IIEXP=KEY(2)
IF (KEY(5).EQ.0) RETURN
PRINT (1,260) ROUTINE,LNCNT
RETURN
100 PRINT*,"SORRY,TURKEY  I CANT CONTINUE"
PRINT(1,150) IIEXP
150 FORMAT (1H0,020)
240 FORMAT (1H0,"IIEXP POSITIVE OVERFLOW  ",A3,"  LINE = ",I5)
260 FORMAT (1H0,"IIEXP NEGATIVE OVERFLOW  ",A3,"  LINE = ",I5)
STOP
END

INTEGER FUNCTION IASGN(I1,ROUTINE,LNCNT,IFNL,KEY,IKEY)
DIMENSION IKEY(4)
DIMENSION KEY(8)
IASGN=I1
IF (LEGVAR(IASGN).NE.0) GO TO 100
IF (IASGN.GT.KEY(1)) GO TO 40
IF (IASGN.LT.KEY(2)) GO TO 60
RETURN
40 IASGN=KEY(1)
IF (KEY(3).EQ.0) RETURN
PRINT (1,240) ROUTINE,LNCNT
RETURN
60 IASGN=KEY(2)
IF (KEY(5).EQ.0) RETURN
PRINT (1,260) ROUTINE,LNCNT
RETURN
100 PRINT*,"SORRY,TURKEY  I CANT CONTINUE"
PRINT(1,150) IASGN
150 FORMAT (1H0,020)
240 FORMAT (1H0,"IASGN POSITIVE OVERFLOW  ",A3,"  LINE = ",I5)
260 FORMAT (1H0,"IASGN NEGATIVE OVERFLOW  ",A3,"  LINE = ",I5)
STOP
END

```

```

REAL FUNCTION RRADD(R1,R2,ROUTINE,LNCNT,IFNL,KEY,TKEY)
DIMENSION KEY(8)
DIMENSION TKEY(4)
RRADD=R1+R2
IF (KEY(3).EQ.1) CALL FOUNDER(RRADD,IFNL,KEY)
IF ((KEY(3).EQ.3).AND.(KEY(4).EQ.2)) CALL TWOTRNC(RRADD,IFNL,KEY)
IF ((KEY(3).EQ.4).AND.(KEY(4).EQ.1)) CALL ONETRNC(RRADD,IFNL,KEY)
IF (RRADD.GT.TKEY(1)) GO TO 20
IF (RRADD.LT.TKEY(2)) GO TO 40
IF ((RRADD.LT.TKEY(3)).AND.(RRADD.GT.0.))
*   GO TO 60
*   IF ((RRADD.GT.TKEY(4)).AND.(RRADD.LT.0.))
*   GO TO 80
RETURN
20 RRADD=TKEY(1)
IF (KEY(5).EQ.0) RETURN
PRINT (1,220) ROUTINE,LNCNT
RETURN
40 RRADD=TKEY(2)
IF (KEY(5).EQ.0) RETURN
PRINT (1,240) ROUTINE,LNCNT
RETURN
60 RRADD=0.0
IF (KEY(5).EQ.0) RETURN
PRINT (1,260) ROUTINE,LNCNT
RETURN
80 RRADD=0.0
IF (KEY(5).EQ.0) RETURN
PRINT (1,280) ROUTINE,LNCNT
RETURN
220 FORMAT (1H0,"RRADD POSITIVE OVERFLOW  ",A8,"  LINE = ",I5)
240 FORMAT (1H0,"RRADD NEGATIVE OVERFLOW  ",A8,"  LINE = ",I5)
260 FORMAT (1H0,"RRADD POSITIVE UNDERFLOW ",A8,"  LINE = ",I5)
280 FORMAT (1H0,"RRADD NEGATIVE UNDERFLOW ",A8,"  LINE = ",I5)
END
REAL FUNCTION RIADD(I1,R2,ROUTINE,LNCNT,IFNL,KEY,TKEY)
DIMENSION KEY(8)
DIMENSION TKEY(4)
RIADD=I1+R2
IF (KEY(3).EQ.1) CALL FOUNDER(RIADD,IFNL,KEY)
IF ((KEY(3).EQ.3).AND.(KEY(4).EQ.2)) CALL TWOTRNC(RIADD,IFNL,KEY)
IF ((KEY(3).EQ.4).AND.(KEY(4).EQ.1)) CALL ONETRNC(RIADD,IFNL,KEY)
IF (RIADD.GT.TKEY(1)) GO TO 20
IF (RIADD.LT.TKEY(2)) GO TO 40
IF ((RIADD.LT.TKEY(3)).AND.(RIADD.GT.0.))
*   GO TO 60
*   IF ((RIADD.GT.TKEY(4)).AND.(RIADD.LT.0.))
*   GO TO 80
RETURN
20 RIADD=TKEY(1)
IF (KEY(5).EQ.0) RETURN
PRINT (1,220) ROUTINE,LNCNT
RETURN
40 RIADD=TKEY(2)
IF (KEY(5).EQ.0) RETURN
PRINT (1,240) ROUTINE,LNCNT
RETURN
60 RIADD=0.0
IF (KEY(5).EQ.0) RETURN
PRINT (1,260) ROUTINE,LNCNT
RETURN
80 RIADD=0.0
IF (KEY(5).EQ.0) RETURN
PRINT (1,280) ROUTINE,LNCNT
RETURN
220 FORMAT (1H0,"RIADD POSITIVE OVERFLOW  ",A8,"  LINE = ",I5)
240 FORMAT (1H0,"RIADD NEGATIVE OVERFLOW  ",A8,"  LINE = ",I5)
260 FORMAT (1H0,"RIADD POSITIVE UNDERFLOW ",A8,"  LINE = ",I5)
280 FORMAT (1H0,"RIADD NEGATIVE UNDERFLOW ",A8,"  LINE = ",I5)
END

```

```

REAL FUNCTION R2ADD(R1,I2,ROUTINE,LNCNT,IFNL,KEY,TKEY)
DIMENSION KEY(8)
DIMENSION TKEY(4)
R2ADD=R1+I2
IF (KEY(3).EQ.1) CALL POUNDER(R2ADD,IFNL,KEY)
IF ((KEY(3).EQ.0).AND.(KEY(4).EQ.2)) CALL TWOTENC(R2ADD,IFNL,KEY)
IF ((KEY(3).EQ.0).AND.(KEY(4).EQ.1)) CALL ONETENC(R2ADD,IFNL,KEY)
IF (R2ADD.GT.TKEY(1)) GO TO 20
IF (R2ADD.LT.TKEY(2)) GO TO 40
IF ((R2ADD.LT.TKEY(3)).AND.(R2ADD.GT.0.))
*   GO TO 60
IF ((R2ADD.GT.TKEY(4)).AND.(R2ADD.LT.0.))
*   GO TO 80
RETURN
20 R2ADD=TKEY(1)
IF (KEY(5).EQ.0) RETURN
PRINT (1,220) ROUTINE,LNCNT
RETURN
40 R2ADD=TKEY(2)
IF (KEY(5).EQ.0) RETURN
PRINT (1,240) ROUTINE,LNCNT
RETURN
60 R2ADD=0.0
IF (KEY(5).EQ.0) RETURN
PRINT (1,260) ROUTINE,LNCNT
RETURN
80 R2ADD=0.0
IF (KEY(5).EQ.0) RETURN
PRINT (1,280) ROUTINE,LNCNT
RETURN
220 FORMAT (1H0,"R2ADD POSITIVE OVERFLOW ",A8," LINE = ",I5)
240 FORMAT (1H0,"R2ADD NEGATIVE OVERFLOW ",A8," LINE = ",I5)
260 FORMAT (1H0,"R2ADD POSITIVE UNDERFLOW ",A8," LINE = ",I5)
280 FORMAT (1H0,"R2ADD NEGATIVE UNDERFLOW ",A8," LINE = ",I5)
END

REAL FUNCTION RRMNS(R1,R2,ROUTINE,LNCNT,IFNL,KEY,TKEY)
DIMENSION KEY(8)
DIMENSION TKEY(4)
RRMNS=R1-R2
IF (KEY(3).EQ.1) CALL POUNDER(RRMNS,IFNL,KEY)
IF ((KEY(3).EQ.0).AND.(KEY(4).EQ.2)) CALL TWOTENC(RRMNS,IFNL,KEY)
IF ((KEY(3).EQ.0).AND.(KEY(4).EQ.1)) CALL ONETENC(RRMNS,IFNL,KEY)
IF (RRMNS.GT.TKEY(1)) GO TO 20
IF (RRMNS.LT.TKEY(2)) GO TO 40
IF ((RRMNS.LT.TKEY(3)).AND.(RRMNS.GT.0.))
*   GO TO 60
IF ((RRMNS.GT.TKEY(4)).AND.(RRMNS.LT.0.))
*   GO TO 80
RETURN
20 RRMNS=TKEY(1)
IF (KEY(5).EQ.0) RETURN
PRINT (1,220) ROUTINE,LNCNT
RETURN
40 RRMNS=TKEY(2)
IF (KEY(5).EQ.0) RETURN
PRINT (1,240) ROUTINE,LNCNT
RETURN
60 RRMNS=0.0
IF (KEY(5).EQ.0) RETURN
PRINT (1,260) ROUTINE,LNCNT
RETURN
80 RRMNS=0.0
IF (KEY(5).EQ.0) RETURN
PRINT (1,280) ROUTINE,LNCNT
RETURN
220 FORMAT (1H0,"RRMNS POSITIVE OVERFLOW ",A8," LINE = ",I5)
240 FORMAT (1H0,"RRMNS NEGATIVE OVERFLOW ",A8," LINE = ",I5)
260 FORMAT (1H0,"RRMNS POSITIVE UNDERFLOW ",A8," LINE = ",I5)
280 FORMAT (1H0,"RRMNS NEGATIVE UNDERFLOW ",A8," LINE = ",I5)
END

```

```

REAL FUNCTION R1MNS(I1,R2,ROUTINE,LNCNT,IFNL,KEY,TKEY)
DIMENSION KEY(8)
DIMENSION TKEY(4)
R1MNS=I1-R2
IF (KEY(3).EQ.1) CALL FOUNDER(R1MNS,IFNL,KEY)
IF ((KEY(3).EQ.1).AND.(KEY(4).EQ.2)) CALL TNOTEND(R1MNS,IFNL,KEY)
IF ((KEY(3).EQ.1).AND.(KEY(4).EQ.1)) CALL ONETEND(R1MNS,IFNL,KEY)
IF (R1MNS.GT.TKEY(1)) GO TO 20
IF (R1MNS.LT.TKEY(2)) GO TO 40
IF ((R1MNS.LT.TKEY(3)).AND.(R1MNS.GT.0.))
*   GO TO 60
IF ((R1MNS.GT.TKEY(4)).AND.(R1MNS.LT.0.))
*   GO TO 80
RETURN
20 R1MNS=TKEY(1)
IF (KEY(5).EQ.0) RETURN
PRINT (1,220) ROUTINE,LNCNT
RETURN
40 R1MNS=TKEY(2)
IF (KEY(5).EQ.0) RETURN
PRINT (1,240) ROUTINE,LNCNT
RETURN
60 R1MNS=0.0
IF (KEY(5).EQ.1) RETURN
PRINT (1,260) ROUTINE,LNCNT
RETURN
80 R1MNS=0.0
IF (KEY(5).EQ.0) RETURN
PRINT (1,280) ROUTINE,LNCNT
RETURN
220 FORMAT (1H0,"R1MNS POSITIVE OVERFLOW ",A8," LINE = ",I5)
240 FORMAT (1H0,"R1MNS NEGATIVE OVERFLOW ",A8," LINE = ",I5)
260 FORMAT (1H0,"R1MNS POSITIVE UNDERFLOW ",A8," LINE = ",I5)
280 FORMAT (1H0,"R1MNS NEGATIVE UNDERFLOW ",A8," LINE = ",I5)
END

REAL FUNCTION R2MNS(I1,I2,ROUTINE,LNCNT,IFNL,KEY,TKEY)
DIMENSION KEY(8)
DIMENSION TKEY(4)
R2MNS=I1-I2
IF (KEY(3).EQ.1) CALL FOUNDER(R2MNS,IFNL,KEY)
IF ((KEY(3).EQ.1).AND.(KEY(4).EQ.2)) CALL TNOTEND(R2MNS,IFNL,KEY)
IF ((KEY(3).EQ.1).AND.(KEY(4).EQ.1)) CALL ONETEND(R2MNS,IFNL,KEY)
IF (R2MNS.GT.TKEY(1)) GO TO 20
IF (R2MNS.LT.TKEY(2)) GO TO 40
IF ((R2MNS.LT.TKEY(3)).AND.(R2MNS.GT.0.))
*   GO TO 60
IF ((R2MNS.GT.TKEY(4)).AND.(R2MNS.LT.0.))
*   GO TO 80
RETURN
20 R2MNS=TKEY(1)
IF (KEY(5).EQ.0) RETURN
PRINT (1,220) ROUTINE,LNCNT
RETURN
40 R2MNS=TKEY(2)
IF (KEY(5).EQ.0) RETURN
PRINT (1,240) ROUTINE,LNCNT
RETURN
60 R2MNS=0.0
IF (KEY(5).EQ.0) RETURN
PRINT (1,260) ROUTINE,LNCNT
RETURN
80 R2MNS=0.0
IF (KEY(5).EQ.0) RETURN
PRINT (1,280) ROUTINE,LNCNT
RETURN
220 FORMAT (1H0,"R2MNS POSITIVE OVERFLOW ",A8," LINE = ",I5)
240 FORMAT (1H0,"R2MNS NEGATIVE OVERFLOW ",A8," LINE = ",I5)
260 FORMAT (1H0,"R2MNS POSITIVE UNDERFLOW ",A8," LINE = ",I5)
280 FORMAT (1H0,"R2MNS NEGATIVE UNDERFLOW ",A8," LINE = ",I5)
END

```

```

REAL FUNCTION RRMPY(R1,R2,ROUTINE,LNCNT,IFNL,KEY,TKEY)
DIMENSION KEY(8)
DIMENSION TKEY(4)
RRMPY=R1-R2
IF (KEY(3).EQ.1) CALL FOUNDER(RRMPY,IFNL,KEY)
IF ((KEY(3).EQ.0).AND.(KEY(4).EQ.2)) CALL TWOTENC(RRMPY,IFNL,KEY)
IF ((KEY(3).EQ.0).AND.(KEY(4).EQ.1)) CALL ONETENC(RRMPY,IFNL,KEY)
IF (RRMPY.GT.TKEY(1)) GO TO 20
IF (RRMPY.LT.TKEY(2)) GO TO 40
IF ((RRMPY.LT.TKEY(3)).AND.(RRMPY.GE.0.))
*   GO TO 60
IF ((RRMPY.GT.TKEY(4)).AND.(RRMPY.LT.0.))
*   GO TO 80
RETURN
20 RRMPY=TKEY(1)
IF (KEY(5).EQ.0) RETURN
PRINT (1,220) ROUTINE,LNCNT
RETURN
40 RRMPY=TKEY(2)
IF (KEY(5).EQ.0) RETURN
PRINT (1,240) ROUTINE,LNCNT
RETURN
60 RRMPY=0.0
IF (KEY(5).EQ.0) RETURN
PRINT (1,260) ROUTINE,LNCNT
RETURN
80 RRMPY=0.0
IF (KEY(5).EQ.0) RETURN
PRINT (1,280) ROUTINE,LNCNT
RETURN
220 FORMAT (1H0,"RRMPY POSITIVE OVERFLOW  ",A8,"  LINE = ",I5)
240 FORMAT (1H0,"RRMPY NEGATIVE OVERFLOW  ",A8,"  LINE = ",I5)
260 FORMAT (1H0,"RRMPY POSITIVE UNDERFLOW  ",A8,"  LINE = ",I5)
280 FORMAT (1H0,"RRMPY NEGATIVE UNDERFLOW  ",A8,"  LINE = ",I5)
END

REAL FUNCTION R1MPY(I1,R2,ROUTINE,LNCNT,IFNL,KEY,TKEY)
DIMENSION KEY(8)
DIMENSION TKEY(4)
R1MPY=I1-R2
IF (KEY(3).EQ.1) CALL FOUNDER(R1MPY,IFNL,KEY)
IF ((KEY(3).EQ.0).AND.(KEY(4).EQ.2)) CALL TWOTENC(R1MPY,IFNL,KEY)
IF ((KEY(3).EQ.0).AND.(KEY(4).EQ.1)) CALL ONETENC(R1MPY,IFNL,KEY)
IF (R1MPY.GT.TKEY(1)) GO TO 20
IF (R1MPY.LT.TKEY(2)) GO TO 40
IF ((R1MPY.LT.TKEY(3)).AND.(R1MPY.GE.0.))
*   GO TO 60
IF ((R1MPY.GT.TKEY(4)).AND.(R1MPY.LT.0.))
*   GO TO 80
RETURN
20 R1MPY=TKEY(1)
IF (KEY(5).EQ.0) RETURN
PRINT (1,220) ROUTINE,LNCNT
RETURN
40 R1MPY=TKEY(2)
IF (KEY(5).EQ.0) RETURN
PRINT (1,240) ROUTINE,LNCNT
RETURN
60 R1MPY=0.0
IF (KEY(5).EQ.0) RETURN
PRINT (1,260) ROUTINE,LNCNT
RETURN
80 R1MPY=0.0
IF (KEY(5).EQ.0) RETURN
PRINT (1,280) ROUTINE,LNCNT
RETURN
220 FORMAT (1H0,"R1MPY POSITIVE OVERFLOW  ",A8,"  LINE = ",I5)
240 FORMAT (1H0,"R1MPY NEGATIVE OVERFLOW  ",A8,"  LINE = ",I5)
260 FORMAT (1H0,"R1MPY POSITIVE UNDERFLOW  ",A8,"  LINE = ",I5)
280 FORMAT (1H0,"R1MPY NEGATIVE UNDERFLOW  ",A8,"  LINE = ",I5)
END

```

```

REAL FUNCTION R2MPY(R1,I2,ROUTINE,LNCNT,IFNL,KEY,TKEY)
DIMENSION KEY(5)
DIMENSION TKEY(4)
R2MPY=R1*I2
IF (KEY(3).EQ.1) CALL FOUNDER(R2MPY,IFNL,KEY)
IF ((KEY(3).EQ.1).AND.(KEY(4).EQ.2)) CALL TWOTENC(R2MPY,IFNL,KEY)
IF ((KEY(3).EQ.1).AND.(KEY(4).EQ.1)) CALL ONETENC(R2MPY,IFNL,KEY)
IF (R2MPY.GT.TKEY(1)) GO TO 20
IF (R2MPY.LT.TKEY(2)) GO TO 40
IF ((R2MPY.LT.TKEY(3)).AND.(R2MPY.GT.0.))
*   GO TO 60
IF ((R2MPY.GT.TKEY(4)).AND.(R2MPY.LT.0.))
*   GO TO 80
RETURN
20 R2MPY=TKEY(1)
IF (KEY(5).EQ.0) RETURN
PRINT (1,220) ROUTINE,LNCNT
RETURN
40 R2MPY=TKEY(2)
IF (KEY(5).EQ.0) RETURN
PRINT (1,240) ROUTINE,LNCNT
RETURN
60 R2MPY=0.0
IF (KEY(5).EQ.0) RETURN
PRINT (1,260) ROUTINE,LNCNT
RETURN
80 R2MPY=0.0
IF (KEY(5).EQ.0) RETURN
PRINT (1,280) ROUTINE,LNCNT
RETURN
220 FORMAT (1H0,"R2MPY POSITIVE OVERFLOW ",A5," LINE = ",I5)
240 FORMAT (1H0,"R2MPY NEGATIVE OVERFLOW ",A5," LINE = ",I5)
260 FORMAT (1H0,"R2MPY POSITIVE UNDERFLOW ",A5," LINE = ",I5)
280 FORMAT (1H0,"R2MPY NEGATIVE UNDERFLOW ",A5," LINE = ",I5)
END

REAL FUNCTION RRPVD(R1,R2,ROUTINE,LNCNT,IFNL,KEY,TKEY)
DIMENSION KEY(5)
DIMENSION TKEY(4)
RRPVD=R1/R2
IF (KEY(3).EQ.1) CALL FOUNDER(RRPVD,IFNL,KEY)
IF ((KEY(3).EQ.1).AND.(KEY(4).EQ.2)) CALL TWOTENC(RRPVD,IFNL,KEY)
IF ((KEY(3).EQ.1).AND.(KEY(4).EQ.1)) CALL ONETENC(RRPVD,IFNL,KEY)
IF (RRPVD.GT.TKEY(1)) GO TO 20
IF (RRPVD.LT.TKEY(2)) GO TO 40
IF ((RRPVD.LT.TKEY(3)).AND.(RRPVD.GT.0.))
*   GO TO 60
IF ((RRPVD.GT.TKEY(4)).AND.(RRPVD.LT.0.))
*   GO TO 80
RETURN
20 RRPVD=TKEY(1)
IF (KEY(5).EQ.0) RETURN
PRINT (1,220) ROUTINE,LNCNT
RETURN
40 RRPVD=TKEY(2)
IF (KEY(5).EQ.0) RETURN
PRINT (1,240) ROUTINE,LNCNT
RETURN
60 RRPVD=0.0
IF (KEY(5).EQ.0) RETURN
PRINT (1,260) ROUTINE,LNCNT
RETURN
80 RRPVD=0.0
IF (KEY(5).EQ.0) RETURN
PRINT (1,280) ROUTINE,LNCNT
RETURN
220 FORMAT (1H0,"RRPVD POSITIVE OVERFLOW ",A5," LINE = ",I5)
240 FORMAT (1H0,"RRPVD NEGATIVE OVERFLOW ",A5," LINE = ",I5)
260 FORMAT (1H0,"RRPVD POSITIVE UNDERFLOW ",A5," LINE = ",I5)
280 FORMAT (1H0,"RRPVD NEGATIVE UNDERFLOW ",A5," LINE = ",I5)
END

```

```

REAL FUNCTION R1DVD(I1,R2,ROUTINE,LNCNT,IFNL,KEY,TKEY)
DIMENSION KEY(8)
DIMENSION TKEY(4)
R1DVD=I1/R2
IF (KEY(3).EQ.1) CALL FOUNDER(R1DVD,IFNL,KEY)
IF ((KEY(3).EQ.3).AND.(KEY(4).EQ.2)) CALL TWOTRNC(R1DVD,IFNL,KEY)
IF ((KEY(3).EQ.3).AND.(KEY(4).EQ.1)) CALL ONETRNC(R1DVD,IFNL,KEY)
IF (R1DVD.GT.TKEY(1)) GO TO 20
IF (R1DVD.LT.TKEY(2)) GO TO 40
IF ((R1DVD.LT.TKEY(3)).AND.(R1DVD.GT.0.))
*   GO TO 60
IF ((R1DVD.GT.TKEY(4)).AND.(R1DVD.LT.0.))
*   GO TO 80
RETURN
20 R1DVD=TKEY(1)
IF (KEY(5).EQ.0) RETURN
PRINT (1,220) ROUTINE,LNCNT
RETURN
40 R1DVD=TKEY(2)
IF (KEY(5).EQ.0) RETURN
PRINT (1,240) ROUTINE,LNCNT
RETURN
60 R1DVD=0.0
IF (KEY(5).EQ.0) RETURN
PRINT (1,260) ROUTINE,LNCNT
RETURN
80 R1DVD=0.0
IF (KEY(5).EQ.0) RETURN
PRINT (1,280) ROUTINE,LNCNT
RETURN
220 FORMAT (1H0,"R1DVD POSITIVE OVERFLOW ",A8," LINE = ",I5)
240 FORMAT (1H0,"R1DVD NEGATIVE OVERFLOW ",A8," LINE = ",I5)
260 FORMAT (1H0,"R1DVD POSITIVE UNDERFLOW ",A8," LINE = ",I5)
280 FORMAT (1H0,"R1DVD NEGATIVE UNDERFLOW ",A8," LINE = ",I5)
END
REAL FUNCTION R2DVD(R1,I2,ROUTINE,LNCNT,IFNL,KEY,TKEY)
DIMENSION KEY(8)
DIMENSION TKEY(4)
R2DVD=R1/I2
IF (KEY(3).EQ.1) CALL FOUNDER(R2DVD,IFNL,KEY)
IF ((KEY(3).EQ.3).AND.(KEY(4).EQ.2)) CALL TWOTRNC(R2DVD,IFNL,KEY)
IF ((KEY(3).EQ.3).AND.(KEY(4).EQ.1)) CALL ONETRNC(R2DVD,IFNL,KEY)
IF (R2DVD.GT.TKEY(1)) GO TO 20
IF (R2DVD.LT.TKEY(2)) GO TO 40
IF ((R2DVD.LT.TKEY(3)).AND.(R2DVD.GT.0.))
*   GO TO 60
IF ((R2DVD.GT.TKEY(4)).AND.(R2DVD.LT.0.))
*   GO TO 80
RETURN
20 R2DVD=TKEY(1)
IF (KEY(5).EQ.0) RETURN
PRINT (1,220) ROUTINE,LNCNT
RETURN
40 R2DVD=TKEY(2)
IF (KEY(5).EQ.0) RETURN
PRINT (1,240) ROUTINE,LNCNT
RETURN
60 R2DVD=0.0
IF (KEY(5).EQ.0) RETURN
PRINT (1,260) ROUTINE,LNCNT
RETURN
80 R2DVD=0.0
IF (KEY(5).EQ.0) RETURN
PRINT (1,280) ROUTINE,LNCNT
RETURN
220 FORMAT (1H0,"R2DVD POSITIVE OVERFLOW ",A8," LINE = ",I5)
240 FORMAT (1H0,"R2DVD NEGATIVE OVERFLOW ",A8," LINE = ",I5)
260 FORMAT (1H0,"R2DVD POSITIVE UNDERFLOW ",A8," LINE = ",I5)
280 FORMAT (1H0,"R2DVD NEGATIVE UNDERFLOW ",A8," LINE = ",I5)
END

```

```

PEAL FUNCTION RREXP(R1,R2,ROUTINE,LNCNT,IFNL,KEY,TKEY)
DIMENSION KEY(6)
DIMENSION TKEY(4)
RREXP=R1-R2
IF (KEY(3).EQ.1) CALL FOUNDER(RREXP,IFNL,KEY)
IF ((KEY(3).EQ.0).AND.(KEY(4).EQ.2)) CALL TWOTENC(RREXP,IFNL,KEY)
IF ((KEY(3).EQ.0).AND.(KEY(4).EQ.1)) CALL ONETENC(RREXP,IFNL,KEY)
IF (RREXP.GT.TKEY(1)) GO TO 20
IF (RREXP.LT.TKEY(2)) GO TO 40
IF ((RREXP.LT.TKEY(3)).AND.(RREXP.GT.0.))
*   GO TO 60
IF ((RREXP.GT.TKEY(4)).AND.(RREXP.LT.0.))
*   GO TO 80
RETURN
20 RREXP=TKEY(1)
IF (KEY(5).EQ.0) RETURN
PRINT (1,220) ROUTINE,LNCNT
RETURN
40 RREXP=TKEY(2)
IF (KEY(5).EQ.0) RETURN
PRINT (1,240) ROUTINE,LNCNT
RETURN
60 RREXP=0.0
IF (KEY(5).EQ.0) RETURN
PRINT (1,260) ROUTINE,LNCNT
RETURN
80 RREXP=0.0
IF (KEY(5).EQ.0) RETURN
PRINT (1,280) ROUTINE,LNCNT
RETURN
220 FORMAT (1H0,"RREXP POSITIVE OVERFLOW ",A8," LINE = ",I5)
240 FORMAT (1H0,"RREXP NEGATIVE OVERFLOW ",A8," LINE = ",I5)
260 FORMAT (1H0,"RREXP POSITIVE UNDERFLOW ",A8," LINE = ",I5)
280 FORMAT (1H0,"RREXP NEGATIVE UNDERFLOW ",A8," LINE = ",I5)
END
PEAL FUNCTION R1EXP(R1,R2,ROUTINE,LNCNT,IFNL,KEY,TKEY)
DIMENSION KEY(6)
DIMENSION TKEY(4)
R1EXP=R1-R2
IF (KEY(3).EQ.1) CALL FOUNDER(R1EXP,IFNL,KEY)
IF ((KEY(3).EQ.0).AND.(KEY(4).EQ.2)) CALL TWOTENC(R1EXP,IFNL,KEY)
IF ((KEY(3).EQ.0).AND.(KEY(4).EQ.1)) CALL ONETENC(R1EXP,IFNL,KEY)
IF (R1EXP.GT.TKEY(1)) GO TO 20
IF (R1EXP.LT.TKEY(2)) GO TO 40
IF ((R1EXP.LT.TKEY(3)).AND.(R1EXP.GT.0.))
*   GO TO 60
IF ((R1EXP.GT.TKEY(4)).AND.(R1EXP.LT.0.))
*   GO TO 80
RETURN
20 R1EXP=TKEY(1)
IF (KEY(5).EQ.0) RETURN
PRINT (1,220) ROUTINE,LNCNT
RETURN
40 R1EXP=TKEY(2)
IF (KEY(5).EQ.0) RETURN
PRINT (1,240) ROUTINE,LNCNT
RETURN
60 R1EXP=0.0
IF (KEY(5).EQ.0) RETURN
PRINT (1,260) ROUTINE,LNCNT
RETURN
80 R1EXP=0.0
IF (KEY(5).EQ.0) RETURN
PRINT (1,280) ROUTINE,LNCNT
RETURN
220 FORMAT (1H0,"R1EXP POSITIVE OVERFLOW ",A8," LINE = ",I5)
240 FORMAT (1H0,"R1EXP NEGATIVE OVERFLOW ",A8," LINE = ",I5)
260 FORMAT (1H0,"R1EXP POSITIVE UNDERFLOW ",A8," LINE = ",I5)
280 FORMAT (1H0,"R1EXP NEGATIVE UNDERFLOW ",A8," LINE = ",I5)
END

```

```

REAL FUNCTION R2EXP(R1, I2, ROUTINE, LNCNT, IFNL, KEY, TKEY)
DIMENSION KEY(8)
DIMENSION TKEY(4)
R2EXP=R1*I2
IF (KEY(3).EQ.1) CALL ROUNDER(R2EXP, IFNL, KEY)
IF ((KEY(3).EQ.0).AND.(KEY(4).EQ.2)) CALL THOTEND(R2EXP, IFNL, KEY)
IF ((KEY(3).EQ.1).AND.(KEY(4).EQ.1)) CALL ONETEND(R2EXP, IFNL, KEY)
IF (R2EXP.GT.TKEY(1)) GO TO 20
IF (R2EXP.LT.TKEY(2)) GO TO 40
IF ((R2EXP.LT.TKEY(3)).AND.(R2EXP.GT.0.))
*   GO TO 60
IF ((R2EXP.GT.TKEY(4)).AND.(R2EXP.LT.0.))
*   GO TO 80
RETURN
20 R2EXP=TKEY(1)
IF (KEY(5).EQ.0) RETURN
PRINT (1, 220) ROUTINE, LNCNT
RETURN
40 R2EXP=TKEY(2)
IF (KEY(5).EQ.0) RETURN
PRINT (1, 240) ROUTINE, LNCNT
RETURN
60 R2EXP=0.0
IF (KEY(5).EQ.0) RETURN
PRINT (1, 260) ROUTINE, LNCNT
RETURN
80 R2EXP=0.0
IF (KEY(5).EQ.0) RETURN
PRINT (1, 280) ROUTINE, LNCNT
RETURN
220 FORMAT (1H0, "R2EXP POSITIVE OVERFLOW ", A3, " LINE = ", I5)
240 FORMAT (1H0, "R2EXP NEGATIVE OVERFLOW ", A3, " LINE = ", I5)
260 FORMAT (1H0, "R2EXP POSITIVE UNDERFLOW ", A3, " LINE = ", I5)
280 FORMAT (1H0, "R2EXP NEGATIVE UNDERFLOW ", A3, " LINE = ", I5)
END
REAL FUNCTION RASGN(R1, ROUTINE, LNCNT, IFNL, KEY, TKEY)
DIMENSION KEY(8)
DIMENSION TKEY(4)
RASGN=R1
IF (KEY(3).EQ.1) CALL ROUNDER(RASGN, IFNL, KEY)
IF ((KEY(3).EQ.1).AND.(KEY(4).EQ.2)) CALL THOTEND(RASGN, IFNL, KEY)
IF ((KEY(3).EQ.0).AND.(KEY(4).EQ.1)) CALL ONETEND(RASGN, IFNL, KEY)
IF (RASGN.GT.TKEY(1)) GO TO 20
IF (RASGN.LT.TKEY(2)) GO TO 40
IF ((RASGN.LT.TKEY(3)).AND.(RASGN.GT.0.))
*   GO TO 60
IF ((RASGN.GT.TKEY(4)).AND.(RASGN.LT.0.))
*   GO TO 80
RETURN
20 RASGN=TKEY(1)
IF (KEY(5).EQ.0) RETURN
PRINT (1, 220) ROUTINE, LNCNT
RETURN
40 RASGN=TKEY(2)
IF (KEY(5).EQ.0) RETURN
PRINT (1, 240) ROUTINE, LNCNT
RETURN
60 RASGN=0.0
IF (KEY(5).EQ.0) RETURN
PRINT (1, 260) ROUTINE, LNCNT
RETURN
80 RASGN=0.0
IF (KEY(5).EQ.0) RETURN
PRINT (1, 280) ROUTINE, LNCNT
RETURN
220 FORMAT (1H0, "RASGN POSITIVE OVERFLOW ", A3, " LINE = ", I5)
240 FORMAT (1H0, "RASGN NEGATIVE OVERFLOW ", A3, " LINE = ", I5)
260 FORMAT (1H0, "RASGN POSITIVE UNDERFLOW ", A3, " LINE = ", I5)
280 FORMAT (1H0, "RASGN NEGATIVE UNDERFLOW ", A3, " LINE = ", I5)
END

```

Appendix C

Mathematical Function Approximations

In chapter 4, routines were presented which were "optimal" for the AN/AYK-15A flight computer in the sense that they provided the required accuracy specified by reference 57 while requiring as little execution time as possible. The routines presented in this appendix were also analyzed and were deemed to be inferior to those presented in chapter 4 for the AN/AYK-15A digital processor. For other applications, however, the routines presented in chapter 4 may not be satisfactory. Routines for approximating the square root, sine, and cosine functions are presented in this appendix, along with some additional references.

Sine Approximations

For the sine function, Taylor series, minimax, polynomial fraction, continued fraction, and Chebyshev expansion approximations were tested. Each proposed solution was executed using the n-bit simulator with a 23-bit mantissa (single precision for the AN/AYK-15A) being specified. Where applicable, Horner's Rule for nested multiplications was applied in evaluating polynomials, even though solutions are not shown as such.

Sine Solution 1. This solution is a truncated Taylor series approximation of degree 7 and is computed by

$$\sin(X) = AX + BX^3 + CX^5 + DX^7 \quad X \in [0, \pi/2] \quad (C-1)$$

with the coefficients being shown in column 2 of Table C-1, which follows the sine solutions. This approximation was not evaluated as a possible candidate for use in the F-16 software, but rather because it appears in software being tested for use in the A-10 aircraft.

Sine Solution 2. This solution is a truncated Taylor series

approximation of degree 11 which was reduced to degree 9 by power series economization (Ref 72:75). This approximation is computed by

$$\sin(\pi X/2) = AX + BX^3 + CX^5 + DX^7 + EX^9 \quad X \in [0,1] \quad (C-2)$$

with the coefficients being shown in column 3 of Table C-1. This approximation was discarded because 1) it did not minimize either the maximum relative or absolute error, and 2) it required more multiplications and additions than were required to meet the accuracy specifications.

Sine Solution 3. This solution is a truncated Taylor series approximation of degree 13 which was reduced to degree 7 by three power series economizations. This approximation is computed by

$$\sin(\pi X/2) = AX + BX^3 + CX^5 + DX^7 \quad X \in [0,1] \quad (C-3)$$

with the coefficients being shown in column 4 of Table C-1. This approximation was discarded because the maximum absolute and relative error bounds were greater than those of the minimax approximation presented in chapter 4.

Sine Solution 4. This solution (Ref 72:82) is a rational approximation of the form

$$\sin(\pi X/2) = \frac{AX + BX^3}{1 + CX^2} \quad X \in [0,1] \quad (C-4)$$

where

$$A = \pi/2$$

$$B = \left(\frac{-7}{60}\right) \left(\frac{\pi^2}{2}\right)$$

$$C = \left(\frac{1}{20}\right) \left(\frac{\pi^2}{2}\right)$$

The values for these coefficients are shown in column 5 of Table C-1.

Since this solution was obtained from a truncated fifth-order Taylor series, the approximate truncation error e_t is

$$e_t = \frac{\left(\frac{\pi}{2}\right)^7 \left(\frac{1}{7!}\right) x^7}{1 + \frac{1}{20} \left(\frac{\pi}{2}\right)^2 x^2} = \frac{0.0046815 x^7}{1 + 0.12337 x^2} \quad (C-5)$$

This approximation was evaluated to be used in comparing results obtained from continued fraction representations. Since a fifth-order Taylor series approximation provided better accuracy than this rational fraction, and this rational fraction provided better accuracy than its representation as a continued fraction, further testing of rational and continued fraction approximations was not conducted.

Sine Solution 5. This solution is the continued fraction representation of the rational fraction presented as solution 4. It was assumed that a floating-point division for the AN/AYK-15A would take approximately 3/2 as long as a floating-point multiplication, so there would be no saving in execution time using a continued fraction over a Taylor series polynomial. This solution (Ref 72:84) is computed as

$$\sin(\pi x/2) = A \left(x - \frac{B}{x + \frac{C}{x}} \right) \quad x \in [0, 1] \quad (C-6)$$

where

$$A = -7\pi/6$$

$$B = \frac{200}{7} \left(\frac{2}{\pi} \right)^2$$

$$C = 20 \left(\frac{2}{\pi} \right)^2$$

The values for these coefficients are shown in column 6 of Table C-1.

This solution had larger absolute and relative errors than either a fifth-order Taylor series or the corresponding rational approximation

presented as solution 4.

Sine Solution 6. This approximation is an expansion in Chebyshev polynomials up to degree 9, resulting in an expression of odd powers of X up to degree 9 (Ref 72:73). This method should be distinguished from Chebyshev approximation which produce minimax polynomials. This approximation using Chebyshev polynomials provides a near-minimax solution for the absolute error and is computed as follows:

$$\sin(\pi X/2) = AX + BX^3 + CX^5 + DX^7 + EX^9 \quad X \in [0,1] \quad (C-7)$$

The values of the coefficients are shown in column 7 of Table C-1. This approximation was not recommended for use with the AN/AYK-15A for two reasons: 1) it did not attempt to minimize the maximum relative error, and 2) since seventh-order polynomials existed which provided sufficient accuracy, the ninth-order approximation with the associated extra addition and multiplication was deemed to require too much execution time.

Sine Solution 7. This solution is a seventh-order minimax approximation which minimizes the maximum absolute error, as opposed to the solution presented in chapter 4 which minimizes the maximum relative error. Since the sine function was used in multiplications and divisions, the polynomial which minimized the maximum relative error was chosen. In other circumstances, this routine might be preferred. This approximation (Ref 71:117,202) is computed as

$$\sin(\pi X/2) = AX + BX^3 + CX^5 + DX^7 \quad X \in [0,1] \quad (C-8)$$

where the values of the coefficients are shown in column 8 of Table C-1.

Sine Solution 8. This solution is a ninth-order minimax approximation which minimizes the maximum relative error. It was tested to determine whether or not more accuracy could be obtained without using

	Sine Solution 1	Sine Solution 2	Sine Solution 3	Sine Solution 4
A	+1.0000000	+1.5707962	+1.570793344	+1.57079633
B	-0.1666667	-0.64596332	-0.645920940	-0.452174868
C	+0.0083333	+0.079688296	+0.079471672	+0.123370055
D	-0.0001984	-0.0046718573	-0.004332878	
E		+0.00015054436		

Table C-1. Sine Coefficients

Sine Solution 5	Sine Solution 6	Sine Solution 7	Sine Solution 8
-3.66519143	+1.5707963	+1.570791011	+1.57079631844
+11.47221432	-0.64596336	-0.6458928493	-0.645963710599
+8.03055026	+0.079688475	+0.0794343442	+0.079689678946
	-0.0046722203	-0.004333095	-0.00467376661
	+0.00015081716		+0.000151485129

Table C-1 (cont). Sine Coefficients

extended precision. This solution (Ref 71:117,204) is computed as

$$\sin(\pi X/2) = AX + BX^3 + CX^5 + DX^7 + EX^9 \quad X \in [0,1] \quad (C-9)$$

where the values of the coefficients are shown in column 9 of Table C-1.

Using this solution, the maximum relative error was less than that produced using the seventh-order polynomial presented in chapter 4.

Machine roundoff error becomes more apparent, however, since the result is only better by a factor of approximately 6, instead of giving two more decimal digits of accuracy as cited by Hart (Ref 71:117).

Cosine Approximation

Only one cosine solution was tested in addition to that presented in chapter 4. This solution (Ref 71:118,207) is an eighth-order mini-max approximation which minimizes the maximum absolute error and is computed by

$$\cos(X) = A + BX^2 + CX^4 + DX^6 + EX^8 \quad X \in [0, \pi/2] \quad (C-10)$$

The values of the coefficients are shown in Table C-2. This routine was not chosen because the relative error for X near $\pi/2$ exceeded 10^{-5} , which was the error bound specified.

A	+0.999999953464
B	-0.499999053455
C	+0.0416635846769
D	-0.0013853704264
E	+0.00002315393167

Table C-2. Cosine Coefficients

Square Root Approximations

The square root approximations differ in the way in which the initial approximation to the square root is obtained. The methods presented use Newton iterations to obtain the desired accuracy. Several

minimax polynomials are given by Hart (Ref 71:94-95) for use in obtaining an initial approximation. Although these approximations were not evaluated in this investigation, they merit consideration also.

Square Root Solution 1. The initial estimate Y_o for this method (Ref 69:31) is accomplished using decimal scaling and a rational function evaluation.

$$Y_o = \frac{1+4X}{X+4} \quad 0.1 \leq X < 10.0 \quad (C-11)$$

Since the AN/AYK-15A computer is a binary machine, this solution was not implemented in a test environment using the n-bit simulator.

Square Root Solution 2. This method (Ref 69:33) uses a Pade approximation and exponent scaling to obtain an initial estimate. As in the method proposed as square root solution 2 in chapter 4, this method shifts the mantissa to obtain an even exponent. The exponent is then divided by two, and the mantissa f is used in the Pade approximation

$$f = A - \frac{B}{f+C} + D/f \quad (C-12)$$

where $|D|$, the relative error term, is less than 0.0014. If $f \in [0.25, 0.50]$, then the coefficients are computed as shown in column 2 of Table C-3, and if $f \in [0.5, 1.0]$, then the coefficients are computed as shown in column 3 of Table C-3. This method requires only one iteration of Newton's method to obtain the desired accuracy (10^{-5}). Since more core storage and logic are required in the initial Pade approximation, the method presented as square root solution 2 in chapter 4 was preferred. Methods such as this one which require only one Newton iteration to obtain a desired accuracy merit further study, however.

	$0.25 \leq f \leq 0.50$	$0.50 < f < 1.00$
A	1.792843	2.535463
B	1.707469	4.829452
C	1.071429	2.142858

Table C-3. Pade Coefficients

Square Root Solution 3. This solution (Ref 56:42) uses two Newton iterations to obtain the required accuracy. The initial approximation Y_0 is obtained using

$$Y_0 = \frac{0.154116 + 1.893872 X}{1.0 + 1.047988 X} \quad 1/16 \leq X < 1 \quad (C-13)$$

with the initial approximation Y_0 having a relative error less than 0.025. The error after two Newton iterations is approximately the same as that using solution 2 presented in chapter 4. With this solution, more effort was expended to obtain the initial approximation, and more coefficients are required to be stored, so the method presented in chapter 4 was preferred.

Square Root Solution 4. This approximation (Ref 70:2-3) for the square root function uses a relatively poor approximation to the square root and then relies on a sufficient number of Newton iterations to obtain the required accuracy. This method has one advantage in that no prestored coefficients are required. The initial approximation Y_0 is obtained for the square root of X by

$$Y_0 = X/2 \quad (C-14)$$

To find the square root of 3000, however, eight Newton iterations are required to yield a relative error less than 10^{-5} . Therefore, this routine was discarded.

Additional References Used in Appendix C

69. Edgar, Elizabeth A. Generation of Elementary Functions Using A Digital Computer. Royal Aircraft Establishment Technical Report 72019, 1972. (AD 903 945).
70. Grove, Wendell E. Brief Numerical Methods. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1966.
71. Hart, J. C., et al. Computer Approximations. Huntington, New York: Robert E. Krieger Publishing Company, 1978.
72. McCracken, Daniel S. and William S. Dorn. Numerical Methods and FORTRAN Programming. New York: John Wiley and Sons, Inc., 1964.

Vita

Richard Arthur Adams, the son of Ralph W. Adams and (the late) Constance J. Adams, was born in New Hampton, Iowa on 12 December 1951, and raised on a farm near Waucoma, Iowa. After graduating from Turkey Valley Community School in 1970, he attended Iowa State University in Ames, Iowa, receiving a Bachelor of Science degree in Mathematics and Computer Science and a commission in the USAF through the AFROTC program. Four months later he entered active duty as a Second Lieutenant and was stationed at Offutt Air Force Base for three years as a computer programmer/analyst. In August 1977, he entered the School of Engineering, Air Force Institute of Technology, to begin graduate studies in Computer Systems.

Permanent Address:

P.O. Box 201

Waucoma

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFIT/GCS/EE/79-1	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Implementation and Testing of Numerical Analysis Techniques in Avionics Applications		5. TYPE OF REPORT & PERIOD COVERED MS Thesis
7. AUTHOR(s) Richard A. Adams Capt USAF		6. PERFORMING ORG. REPORT NUMBER
9. PERFORMING ORGANIZATION NAME AND ADDRESS Air Force Institute of Technology (AFIT-EN) Wright-Patterson AFB, Ohio 45433		8. CONTRACT OR GRANT NUMBER(s)
11. CONTROLLING OFFICE NAME AND ADDRESS Air Force Avionics Laboratory/AAT-2 Wright-Patterson AFB OH 45433		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		12. REPORT DATE March 1979
		13. NUMBER OF PAGES 142
		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES Approved for public release IAW AFR 190-17 J. P. Hipps, Major, USAF Director of Information 16 MAY 1979		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Numerical Analysis Avionics Forward Error Analysis Simulation Computer Modelling		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) Errors due to finite wordlength are unavoidable when aircraft signal processing operations such as flight control, navigation, and fire control are implemented on a digital computer. To reduce these errors to tolerable levels, longer wordlengths can sometimes be employed. The effects of some of the errors, such as those due to arithmetic series truncation, machine roundoff, and quantization of system coefficients, can be lessened somewhat by appropriate numerical analysis techniques.		

DD FORM 1 JAN 73 1473

EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

20. An n-bit simulator which runs on Control Data Corporation (CDC) 6600/CYBER 74 computer systems was modified and then used to evaluate the accuracy of a flight navigation routine coded in FORTRAN. The routines were executed without the simulator to obtain results used for benchmarking. The n-bit simulator was employed to simulate the numerical characteristics of the AN/AYK-15A digital processor. Error plots were constructed which show the maximum errors occurring within small plotting intervals plotted against each individual input value. These plots were used to aid visually in analyzing the error characteristics of the avionics routine as it would be implemented on the AN/AYK-15A.

A critical analysis of the error plots obtained showed that routines which are coded using single-precision floating-point arithmetic are prone to errors which exceed the error bounds specified for the routines. This occurs even though range reductions in the trigonometric function approximations are accomplished using extended precision.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)